

# Linear Dependent Types in a Call-by-Value Scenario

Ugo Dal Lago      Barbara Petit

July 25, 2012

## Abstract

Linear dependent types [8] allow to precisely capture both the extensional behaviour and the time complexity of  $\lambda$ -terms, when the latter are evaluated by Krivine’s abstract machine. In this work, we show that the same paradigm can be applied to call-by-value evaluation. A system of linear dependent types for Plotkin’s PCF is introduced, called  $d\ell PCF_V$ , whose types reflect the complexity of evaluating terms in the so-called CEK machine.  $d\ell PCF_V$  is proved to be sound, but also relatively complete: every true statement about the extensional and intentional behaviour of terms can be derived, provided all true index term inequalities can be used as assumptions.

## 1 Introduction

A variety of methodologies for formally verifying properties of programs have been introduced in the last fifty years. Among them, *type systems* have certain peculiarities. On the one hand, the way one defines a type system makes the task of proving a given program to have a type reasonably simple and modular: a type derivation for a compound program usually consists of some type derivations for the components, appropriately glued together in a syntax-directed way (i.e. attributing a type to a program can usually be done *compositionally*). On the other, the specifications that can be expressed through types have traditionally been weak, although stronger properties have recently become of interest, such as security [23, 22], termination [5], monadic temporal properties [17] or resource bounds [16]. But contrarily to what happens with other formal methods (e.g. model checking or program logics), giving a type to a program  $t$  is a *sound* but *incomplete* way to prove  $t$  to satisfy a specification: there are correct programs which cannot be proved such by way of typing.

In other words, the tension between expressiveness and tractability is particularly evident in the field of type systems, where certain good properties the majority of type systems enjoy (e.g. syntax-directedness) are usually considered as desirable (if not necessary), but also have their drawbacks: some specifications are intrinsically hard to verify locally and compositionally. One specific research field in which the just-described scenario manifests itself is complexity analysis, in which the specification takes the form of concrete or asymptotic bounds on the complexity of the underlying program. Many type systems have been introduced capturing, for instance, the class of polynomial time computable functions [15, 4, 3]. All of them, under mild assumptions, can be employed as tools to certify programs as asymptotically time efficient. However, a tiny slice of the polytime *programs* are generally typable, since the underlying complexity class **FP** is only characterised in a purely extensional sense — for every function in **FP** there is *at least one* typable program computing it.

Gaboardi and the first author have recently introduced [8] a type system for Plotkin’s PCF, called  $d\ell PCF_N$ , in which linearity and a restricted form of dependency in the spirit of Xi’s DML are present:

- **Linearity** makes it possible to finely control the number of times subterms are copied during the evaluation of a term  $t$ , itself a parameter which accurately reflects the time complexity of  $t$  [7].
- **Dependency** allows to type distinct (virtual) copies of a term with distinct types. This gives the type system an extra flexibility similar to that of intersection types.

When mixed together, these two ingredients allow to precisely capture the extensional behaviour of  $\lambda$ -terms *and* the time complexity of their evaluation by Krivine’s abstract machine. Both soundness and relative completeness hold for  $\mathbf{d}\ell\mathbf{PCF}_N$ .

One may argue, however, that the practical relevance of these results is quite limited, given that call-by-name evaluation and KAM are very inefficient: why would one be interested in verifying the complexity of evaluating concrete programs in such a setting?

In this work, we show that linear dependent types can also be applied to the analysis of call-by-value evaluation of functional programs. This is done by introducing another system of linear dependent types for Plotkin’s PCF. The system, called  $\mathbf{d}\ell\mathbf{PCF}_V$ , captures the complexity of evaluating terms by Felleisen and Friedman’s CEK machine [12], a simple abstract machine for call-by-value evaluation.  $\mathbf{d}\ell\mathbf{PCF}_V$  is proved to enjoy the same good properties enjoyed by its sibling  $\mathbf{d}\ell\mathbf{PCF}_N$ , namely soundness and relative completeness: every true statement about the extensional behaviour of terms can be derived, provided all true index term inequalities can be used as assumptions.

Actually,  $\mathbf{d}\ell\mathbf{PCF}_V$  is not merely a variation on  $\mathbf{d}\ell\mathbf{PCF}_N$ : not only typing rules are different, but also the language of types itself must be modified. Roughly,  $\mathbf{d}\ell\mathbf{PCF}_V$  and  $\mathbf{d}\ell\mathbf{PCF}_N$  can be thought as being induced by translations of intuitionistic logic into linear logic: the latter corresponds to Girard’s translation  $A \Rightarrow B \equiv !A \multimap B$ , while the former corresponds to  $A \Rightarrow B \equiv !(A \multimap B)$ . The strong link between translations of IL into ILL and notions of reduction for the  $\lambda$ -calculus is well-known (see *e.g.* [19]) and has been a guide in the design of  $\mathbf{d}\ell\mathbf{PCF}_V$  (this is explained in Section. 2.2).

## 2 Linear Dependent Types, Intuitively

Consider the following program:

$$\mathbf{dbl} = \mathbf{fix} \ f.\lambda x. \mathbf{ifz} \ x \ \mathbf{then} \ x \ \mathbf{else} \ \mathbf{s}(\mathbf{s}(f(\mathbf{p}(x)))).$$

In a type system like PCF [21], the term  $\mathbf{dbl}$  receives type  $\mathbf{Nat} \Rightarrow \mathbf{Nat}$ . As a consequence,  $\mathbf{dbl}$  computes a function on natural numbers without “going wrong”: it takes in input a natural number, and (possibly) produces in output another natural number. The type  $\mathbf{Nat} \Rightarrow \mathbf{Nat}$ , however, does not give any information about *which* specific function on the natural numbers  $\mathbf{dbl}$  computes.

Properties of programs which are completely ignored by ordinary type systems are termination and its most natural refinement, namely termination in *bounded time*. Typing a term  $t$  with  $\mathbf{Nat} \Rightarrow \mathbf{Nat}$  does not guarantee that  $t$ , when applied to a natural number, terminates. Consider, as another example, a slight modification of  $\mathbf{dbl}$ , namely

$$\mathbf{div} = \mathbf{fix} \ f.\lambda x. \mathbf{ifz} \ x \ \mathbf{then} \ x \ \mathbf{else} \ \mathbf{s}(\mathbf{s}(f(x))).$$

It behaves as  $\mathbf{dbl}$  when fed with 0, but it diverges when it receives a positive natural number as an argument. But look:  $\mathbf{div}$  is not so different from  $\mathbf{dbl}$ . Indeed, the second can be obtained from the first by feeding not  $x$  but  $\mathbf{p}(x)$  to  $f$ . And any type system in which  $\mathbf{dbl}$  and  $\mathbf{div}$  are somehow recognised as being fundamentally different must be able to detect the presence of  $\mathbf{p}$  in  $\mathbf{dbl}$  and deduce termination from it. Indeed, sized types [5] and dependent types [24] are able to do so. Going further, we could ask the type system to be able not only to guarantee termination, but also to somehow evaluate the time or space consumption of programs. For example, we could be

interested in knowing that **dbl** takes a polynomial number of steps to be evaluated on any natural number, and actually some type systems able to control the complexity of higher-order programs exist. Good examples are type systems for amortised analysis [16, 14] or those using ideas from linear logic [4, 3]: in all of them, linearity plays a key role.

$\text{d}\ell\text{PCF}_N$  [8] combines some of the ideas presented above with the principles of bounded linear logic (BLL [13]): the cost of evaluating a term is measured by counting how many times function arguments need to be copied during evaluation, and different copies can be given distinct, although uniform, types. Making this information explicit in types permits to compute the cost step by step during the type derivation process. Roughly, typing judgements in  $\text{d}\ell\text{PCF}_N$  are statements like

$$\vdash_{J(a)} t : !_n \text{Nat}[a] \multimap \text{Nat}[I(a)],$$

where  $I$  and  $J$  depend on  $a$  and  $n$  is a natural number capturing the number of times  $t$  uses its argument. But this is not sufficient: analogously to what happens in BLL,  $\text{d}\ell\text{PCF}_N$  makes types more parametric. A type like  $!_n \sigma \multimap \tau$  is replaced by the more parametric type  $!_{a < n} \sigma \multimap \tau$ , which tells us that the argument will be used  $n$  times, and each instance has type  $\sigma$  *where, however* the variable  $a$  is instantiated with a value less than  $n$ . This allows to type each copy of the argument differently but uniformly, since all instances of  $\sigma$  have the same PCF skeleton. This form of *uniform linear dependence* is actually crucial in obtaining the result which makes  $\text{d}\ell\text{PCF}_N$  different from similar type systems, namely completeness. As an example, **dbl** can be typed as follows in  $\text{d}\ell\text{PCF}_N$ :

$$\vdash_a^{\mathcal{E}} \text{dbl} : !_{b < a+1} \text{Nat}[a] \multimap \text{Nat}[2 \times a].$$

This tells us that the argument will be used  $a$  times by **dbl**, namely a number of times equal to its value. And that the cost of evaluation will be itself proportional to  $a$ .

## 2.1 Why Another Type System?

The theory of  $\lambda$ -calculus is full of interesting results, one of them being the so-called Church-Rösser property: both  $\beta$  and  $\beta\eta$  reduction are confluent, *i.e.* if you fire two distinct redexes in a  $\lambda$ -term, you can always “close the diagram” by performing one *or more* rewriting steps. This, however, is not a *local* confluence result, and as such does *not* imply that all reduction strategies are computationally equivalent. Indeed, some of them are normalising (like normal-order evaluation) while some others are not (like innermost reduction). But how about efficiency?

On the one hand, it is well known that optimal reduction *is* indeed possible [18], even if it gives rise to high overheads [1]. On the other, call-by-name can be highly inefficient. Consider, as an example, the composition of **dbl** with itself:

$$\text{dbl2} = \lambda x. \text{dbl}(\text{dbl } x).$$

This takes quadratic time to be evaluated in the KAM: the evaluation of  $(\text{dbl } \underline{n})$  is repeated a linear number of times, whenever it reaches the head position. This actually *can* be seen from within  $\text{d}\ell\text{PCF}_N$ , since

$$\vdash_J^{\mathcal{E}} \text{dbl2} : !_{b < I} \text{Nat}[a] \multimap \text{Nat}[4 \times a].$$

where both  $I$  and  $J$  are quadratic in  $a$ . Call-by-value solves this problem, at the price of not being normalising. Indeed, eager evaluation of **dbl2** when fed with a natural number  $n$  takes linear time in  $n$ . The relative efficiency of call-by-value evaluation, compared to call-by-name, is not a novelty: many modern functional programming languages (like OCaml and Scheme) are based on it, while very few of them evaluate terms in call-by-name order.

For the reasons above, we strongly believe that designing a type system in the style of  $\text{d}\ell\text{PCF}_N$ , but able to deal with eager evaluation, is a step forward applying linear dependent types to actual programming languages.

## 2.2 Call-by-Value, Call-by-Name and Linear Logic

Various notions of evaluation for the  $\lambda$ -calculus can be seen as translations of intuitionistic logic (or of simply-typed  $\lambda$ -calculi) into Girard's linear logic. This correspondence has been investigated in the specific cases of call-by-name (CBN) and call-by-value (CBV) reduction (*e.g.* see the work of Maraist et al. [19]). In this section, we briefly introduce the main ideas behind the correspondence, explaining why linear logic has guided the design of  $\mathbf{d}\ell\mathbf{PCF}_V$ .

The general principle in such translations, is to guarantee that whenever a term *can* possibly be duplicated, it must be mapped to a box in the underlying linear logic proof. In the CBN translation (also called Girard's translation), *any* argument to functions can possibly be substituted for a variable and copied, so arguments are banged during the translation:

$$(A \Rightarrow B)^* = (!A^*) \multimap B^*$$

Adding the quantitative bound on banged types (as explained in the previous section) gives rise to the type  $(!_{a < I}\sigma) \multimap \tau$  for functions (written  $[a < I] \cdot \sigma \multimap \tau$  in [8]). In the same way, *contexts* are banged in the CBN translation: a typing judgement in  $\mathbf{d}\ell\mathbf{PCF}_N$  have the following form:

$$x_1 : !_{a_1 < I_1} \sigma_1, \dots, x_n : !_{a_n < I_n} \sigma_n \vdash_J t : \tau.$$

In the CBV translation,  $\beta$ -reduction should be performed only if the argument is a value. Thus, arguments are not automatically banged during the translation but values are, so that the  $\beta$ -reduction remains blocked until the argument reduces to a value. In the  $\lambda$ -calculus values are functions, hence the translation of the intuitionistic arrow becomes

$$(A \Rightarrow B)^\circ = !(A^\circ \multimap B^\circ).$$

Function types in  $\mathbf{d}\ell\mathbf{PCF}_V$  then become  $!_{a < I}(\sigma \multimap \tau)$ , and a judgement has the form  $x_1 : \sigma_1, \dots, x_n : \sigma_n \vdash_J t : \tau$ . The syntax of types varies fairly much between  $\mathbf{d}\ell\mathbf{PCF}_N$  to  $\mathbf{d}\ell\mathbf{PCF}_V$ , and consequently the two type systems are different, although both of them are greatly inspired by linear logic.

In both cases, however, the “target” of the translation is not the whole of ILL, but rather a restricted version of it, namely BLL, in which the complexity of normalisation is kept under control by shifting from unbounded, infinitary, exponentials to finitary ones. For example, the BLL *contraction* rule allows to merge the first  $I$  copies of  $A$ , and the following  $J$  ones into the first  $I + J$  copies of  $A$ :

$$\frac{\Gamma, !_{a < I} A, !_{a < J} A \{I + a/a\} \vdash B}{\Gamma, x : !_{a < I+J} A \vdash B}$$

We write  $\sigma \uplus \tau = !_{a < I+J} A$  if  $\sigma = !_{a < I} A$  and  $\tau = !_{a < J} A \{I + a/a\}$ . Any time a contraction rule is involved in the CBV translation of a type derivation, a sum  $\uplus$  appears at the same place in the corresponding  $\mathbf{d}\ell\mathbf{PCF}_V$  derivation. Similarly, the *dereliction* rule allows to see any type as the first copy of itself:

$$\frac{\Gamma, A \{0/a\} \vdash B}{\Gamma, !_{a < 1} A \vdash B}$$

hence any dereliction rule appearing in the translation of a typing judgement tells us that the corresponding type is copied once. Both contraction and dereliction appear while typing an application in  $\mathbf{d}\ell\mathbf{PCF}_V$ : the PCF typing rule

$$\frac{\Gamma \vdash t : A \Rightarrow B \quad \Gamma \vdash u : A}{\Gamma \vdash tu : B}$$

corresponds to the following ILL proof:

$$\frac{\frac{\frac{z : A^\circ \multimap B^\circ \vdash z : A^\circ \multimap B^\circ}{!z : !(A^\circ \multimap B^\circ) \vdash z : A^\circ \multimap B^\circ} \text{ der} \quad \Gamma^\circ \vdash t^\circ : !(A^\circ \multimap B^\circ)}{\Gamma^\circ \vdash t^\circ : A^\circ \multimap B^\circ} \quad \Gamma^\circ \vdash u^\circ : A^\circ}{(\Gamma^\circ = !\Gamma') \quad \frac{\Gamma^\circ, \Gamma^\circ \vdash t^\circ u^\circ : B^\circ}{\Gamma^\circ \vdash t^\circ u^\circ : B^\circ} \text{ contr}} \text{ contr}$$

which becomes the following, when appropriately decorated according to the principles of BLL (writing  $A_0$  and  $B_0$  for  $A\{0/a\}$  and  $B\{0/a\}$ ):

$$\frac{\frac{z: A_0^\circ \multimap B_0^\circ \vdash z: A_0^\circ \multimap B_0^\circ}{!z: !_{a<1}(A^\circ \multimap B^\circ) \vdash z: A_0^\circ \multimap B_0^\circ} \text{ der} \quad \frac{\Gamma^\circ \vdash t^\circ: !_{a<1}(A^\circ \multimap B^\circ)}{\Gamma^\circ \vdash t^\circ: A_0^\circ \multimap B_0^\circ}}{\frac{\Gamma^\circ \vdash t^\circ: A_0^\circ \multimap B_0^\circ \quad \Gamma^\circ \vdash u^\circ: A_0^\circ}{(\Gamma^\circ = !\Gamma') \quad \frac{\Gamma^\circ, \Gamma^\circ \vdash t^\circ u^\circ: B_0^\circ}{\Gamma^\circ \uplus \Gamma^\circ \vdash t^\circ u^\circ: B_0^\circ} \text{ contr}}} \text{ This CBV translation of the application rule hence leads to the typing rule for applications in } \mathbf{d\ell PCF_V}:$$

This CBV translation of the application rule hence leads to the typing rule for applications in  $\mathbf{d\ell PCF_V}$ :

$$\frac{\Gamma \vdash_K t : !_{a<1}(\sigma \multimap \tau) \quad \Delta \vdash_H u : \sigma\{0/a\}}{\Gamma \uplus \Delta \vdash_{K+H} tu : \tau\{0/a\}}$$

The same kind of analysis enables to derive the typing rule for abstractions (whose call-by-value translation requires the use of a promotion rule) in  $\mathbf{d\ell PCF_V}$ :

$$\frac{\Gamma, x : \sigma \vdash_K t : \tau}{\sum_{a<I} \Gamma \vdash_{I+\sum_{a<I} K} \lambda x. t : !_{a<I}(\sigma \multimap \tau)}$$

One may wonder what  $I$  represents in this typing rule, and more generally in a judgement such as

$$\Gamma \vdash_K t : !_{a<I} A.$$

This is actually the main new idea of  $\mathbf{d\ell PCF_V}$ : such a judgement intuitively means that the value to which  $t$  reduces will be used  $I$  times by the environment. If  $t$  is applied to an argument  $u$ , then  $t$  must reduce to an abstraction  $\lambda x. s$ , that is destructured by the argument without being duplicated. In that case,  $I = 1$ , as indicated by the application typing rule. On the opposite, if  $t$  is applied to a function  $\lambda x. u$ , then the type of this function must be of the form (up to a substitution of  $b$ )  $!_{b<1}(!_{a<I} A \multimap \tau)$ . This means that  $\lambda x. u$  uses  $I$  times its arguments, or, that  $x$  can appear at most  $I$  times in the reducts of  $u$ .

This suggests that the type derivation of a term is not unique in general: whether a term  $t$  has type  $!_{a<I} A$  or  $!_{a<J} A$  depends on the use we want to make of  $t$ . This intuition will direct us in establishing the typing rules for the other PCF constructs (namely conditional branching and fixpoints).

### 3 $\mathbf{d\ell PCF_V}$ , Formally

In this section, the language of programs and a type system  $\mathbf{d\ell PCF_V}$  for it will be introduced formally. While programs are just terms of a fairly standard  $\lambda$ -calculus (which is very similar to Plotkin's PCF), types may include so-called *index terms*, which are first-order terms denoting natural numbers by which one can express properties about the extensional and intentional behaviour of programs.

#### 3.1 Index Terms and Equational Programs

Syntactically, index terms are built either from function symbols from a given untyped signature  $\Theta$  or by applying any of two special term constructs:

$$I, J, K ::= a \mid f(I_1, \dots, I_n) \mid \sum_{a<I} J \mid \bigtriangleup_a^{I,J} K.$$

Here,  $f$  is a symbol of arity  $n$  from  $\Theta$  and  $a$  is a variable drawn from a set  $\mathcal{V}$  of *index variables*. We assume the symbols  $0, 1$  (with arity 0) and  $+, -$  (with arity 2) are always part of  $\Theta$ . An index term in the form  $\sum_{a<I} J$  is a *bounded sum*, while one in the form  $\bigtriangleup_a^{I,J} K$  is a *forest cardinality*. For every natural number  $n$ , the index term  $n$  is just  $\underbrace{1 + 1 + \dots + 1}_{n \text{ times}}$ .

Index terms are meant to denote natural numbers, possibly depending on the (unknown) values of variables. Variables can be instantiated with other index terms, *e.g.*  $I\{J/a\}$ . So, index terms can also act as first order functions. What is the meaning of the function symbols from  $\Theta$ ? It is the one induced by an equational program  $\mathcal{E}$ . Formally, an *equational program*  $\mathcal{E}$  over a signature  $\Theta$  is a set of equations in the form  $I = J$  where both  $I$  and  $J$  are index terms. We are interested in equational programs guaranteeing that, whenever symbols in  $\Theta$  are interpreted as partial functions over  $\mathbb{N}$  and  $0, 1, +$  and  $-$  are interpreted in the usual way, the semantics of any function symbol  $f$  can be uniquely determined from  $\mathcal{E}$ . This can be guaranteed by, for example, taking  $\mathcal{E}$  as an Herbrand-Gödel scheme [20] or as an orthogonal constructor term rewriting system [2]. The definition of index terms is parametric on  $\Theta$  and  $\mathcal{E}$ : this way one can tune our type system from a highly undecidable but truly powerful machinery down to a tractable but less expressive formal system.

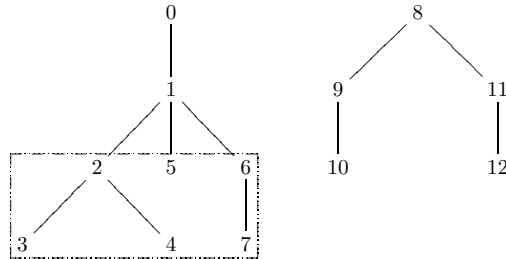
What about the meaning of bounded sums and forest cardinalities? The first is very intuitive: the value of  $\sum_{a < I} J$  is simply the sum of all possible values of  $J$  with  $a$  taking the values from 0 up to  $I$ , excluded. Forest cardinalities, on the other hand, require some effort to be described. Informally,  $\bigtriangleup_a^{I,J} K$  is an index term denoting the number of nodes in a forest composed of  $J$  trees described using  $K$ . All the nodes in the forest are (uniquely) identified by natural numbers. These are obtained by consecutively visiting each tree in pre-order, starting from  $I$ . The term  $K$  has the role of describing the number of children of each forest node, *e.g.* the number of children of the node 0 is  $K\{0/a\}$ . More formally, the meaning of a forest cardinality is defined by the following two equations:

$$\begin{aligned} \bigtriangleup_a^{I,0} K &= 0 \\ \bigtriangleup_a^{I,J+1} K &= \left( \bigtriangleup_a^{I,J} K \right) + 1 + \left( \bigtriangleup_a^{I+1+\bigtriangleup_a^{I,J} K, K\{I+\bigtriangleup_a^{I,J} K/a\}} K \right) \end{aligned}$$

The first equation says that a forest of 0 trees contains no nodes. The second one tells us that a forest of  $J + 1$  trees contains:

- The nodes in the first  $J$  trees;
- plus the nodes in the last tree, which are just one plus the nodes in the immediate subtrees of the root, considered themselves as a forest.

To better understand forest cardinalities, consider the following forest comprising two trees:



It is well described by an index term  $K$  with a free index variable  $a$  such that  $K\{1/a\} = 3$ ;  $K\{n/a\} = 2$  for  $n \in \{2, 8\}$ ;  $K\{n/a\} = 1$  when  $n \in \{0, 6, 9, 11\}$ ; and  $K\{n/a\} = 0$  when  $n \in \{3, 4, 5, 7, 10, 12\}$ . That is,  $K$  describes the number of children of each node. Then  $\bigtriangleup_a^{0,2} K = 13$  since it takes into account the entire forest;  $\bigtriangleup_a^{0,1} K = 8$  since it takes into account only the leftmost tree;  $\bigtriangleup_a^{8,1} K = 5$  since it takes into account only the second tree of the forest; finally,  $\bigtriangleup_a^{2,3} K = 6$  since it takes into account only the three trees (as a forest) within the dashed rectangle.

One may wonder what is the role of forest cardinalities in the type system. Actually, they play a crucial role in the treatment of recursion, where the unfolding of recursion produces a tree-like

$(\lambda x.t) v$	$\rightarrow_v$	$t[x := v]$
$\mathbf{s}(\underline{n})$	$\rightarrow_v$	$\underline{n+1}$
$\mathbf{p}(\underline{n+1})$	$\rightarrow_v$	$\underline{n}$
$\mathbf{p}(\underline{0})$	$\rightarrow_v$	$\underline{0}$
$\text{ifz } \underline{0} \text{ then } t \text{ else } u$	$\rightarrow_v$	$t$
$\text{ifz } \underline{n+1} \text{ then } t \text{ else } u$	$\rightarrow_v$	$u$
$(\text{fix } x.t) v$	$\rightarrow_v$	$(t[x := \text{fix } x.t]) v$

Figure 1: Call-by-value reduction of PCF terms.

structure whose size is just the number of times the (recursively defined) function will be used *globally*. Note that the value of a forest cardinality could also be undefined. For instance, this happens when infinite trees, corresponding to diverging recursive computations, are considered.

The expression  $\llbracket I \rrbracket_\rho^\mathcal{E}$  denotes the meaning of  $I$ , defined by induction along the lines of the previous discussion, where  $\rho : \mathcal{V} \rightarrow \mathbb{N}$  is an assignment and  $\mathcal{E}$  is an equational program giving meaning to the function symbols in  $I$ . Since  $\mathcal{E}$  does not necessarily interpret such symbols as *total* functions, and moreover, the value of a forest cardinality can be undefined,  $\llbracket I \rrbracket_\rho^\mathcal{E}$  can be undefined itself. A *constraint* is an inequality in the form  $I \leq J$ . Such a constraint is *true* (or *satisfied*) in an assignment  $\rho$  if  $\llbracket I \rrbracket_\rho^\mathcal{E}$  and  $\llbracket J \rrbracket_\rho^\mathcal{E}$  are *both* defined and the first is smaller or equal to the latter. Now, for a subset  $\phi$  of  $\mathcal{V}$ , and for a set  $\Phi$  of constraints involving variables in  $\phi$ , the expression

$$\phi; \Phi \models_\mathcal{E} I \leq J$$

denotes the fact that the truth of  $I \leq J$  *semantically* follows from the truth of the constraints in  $\Phi$ . To denote that  $I$  is well defined for  $\mathcal{E}$  and any valuation  $\rho$  satisfying  $\Phi$ , we may write  $\phi; \Phi \models_\mathcal{E} I \Downarrow$  instead of  $\phi; \Phi \models_\mathcal{E} I \leq I$ .

### 3.2 Programs

*Values* and *terms* are generated by the following grammar:

$$\begin{array}{ll} \text{Values:} & v, w ::= \underline{n} \mid \lambda x.t \mid \text{fix } x.t \\ \text{Terms:} & s, t, u ::= x \mid v \mid tu \mid \mathbf{s}(t) \mid \mathbf{p}(t) \\ & \mid \text{ifz } t \text{ then } u \text{ else } s \end{array}$$

Terms can be typed with a well-known type system called PCF: types are those generated by the basic type  $\mathbf{Nat}$  and the binary type constructor  $\Rightarrow$ . Typing rules are standard (see [9]). A notion of (weak) call-by-value reduction  $\rightarrow_v$  can be easily defined: take the rewriting rules in Figure 1 and close them under all applicative contexts. A term  $t$  is said to be a *program* if it can be given the PCF type  $\mathbf{Nat}$  in the empty context. The *multiplicative size*  $\|t\|$  of a term  $t$  is defined as follows:

$$\begin{aligned} \|\underline{n}\| &= \|\lambda x.t\| = \|\text{fix } x.t\| = 0; \\ \|x\| &= 2; \\ \|tu\| &= \|t\| + \|u\| + 2; \\ \|\mathbf{s}(t)\| &= \|t\| + 2; \\ \|\mathbf{p}(t)\| &= \|t\| + 2; \\ \|\text{ifz } t \text{ then } u \text{ else } s\| &= \|t\| + \|u\| + \|s\| + 2. \end{aligned}$$

Notice that the multiplicative size of a term  $t$  is less or equal than its size  $|t|$  (which is defined inductively, similarly to  $\|t\|$ , except for values:  $|\underline{n}| = 2$ , and  $|\text{fix } x.t| = |\lambda x.t| = |t| + 2$ ). Values are not taken into account by the multiplicative size. Indeed, the evaluation of terms (*cf.* Section 3.4) consists first in *scanning* a term until a value is reached (and the cost of this step is measured by the multiplicative size). Then this value is either destructed (*e.g.* when a lambda abstraction is given an argument), either duplicated (*e.g.* when it is itself an argument of a lambda abstraction). The cost of this second step will be measured by the type system  $\mathbf{d}\ell\mathbf{PCF}_V$ .

$\frac{\phi; \Phi \models_{\varepsilon} K \leq I \quad \phi; \Phi \models_{\varepsilon} J \leq H}{\phi; \Phi \vdash_{\varepsilon} \text{Nat}[I, J] \sqsubseteq \text{Nat}[K, H]}$	$\frac{\phi; \Phi \vdash_{\varepsilon} \sigma' \sqsubseteq \sigma \quad \phi; \Phi \vdash_{\varepsilon} \tau \sqsubseteq \tau'}{\phi; \Phi \vdash_{\varepsilon} \sigma \multimap \tau \sqsubseteq \sigma' \multimap \tau'}$	$\frac{(a, \phi); (a < J, \Phi) \vdash_{\varepsilon} A \sqsubseteq B \quad \phi; \Phi \models_{\varepsilon} J \leq I}{\phi; \Phi \vdash_{\varepsilon} [a < I] \cdot A \sqsubseteq [a < J] \cdot B}$
---	---	---

Figure 2: Subtyping derivation rules of  $\text{d}\ell\text{PCF}_V$ .

$\frac{}{\phi; \Phi; \Gamma, x : \sigma \vdash_0^{\varepsilon} x : \sigma} (Ax)$	$\frac{\phi; \Phi; \Gamma \vdash_1^{\varepsilon} t : \sigma \quad \phi; \Phi \vdash_{\varepsilon} \Delta \sqsubseteq \Gamma \quad \phi; \Phi \vdash_{\varepsilon} \sigma \sqsubseteq \tau \quad \phi; \Phi \models_{\varepsilon} I \leq J}{\phi; \Phi; \Delta \vdash_J^{\varepsilon} t : \tau} (Subs)$
$\frac{(a, \phi); (a < I, \Phi); \Gamma, x : \sigma \vdash_K^{\varepsilon} t : \tau}{\phi; \Phi; \sum_{a < I} \Gamma \vdash_{I + \sum_{a < I} K}^{\varepsilon} \lambda x. t : [a < I] \cdot \sigma \multimap \tau} (-\circ)$	$\frac{\phi; \Phi; \Gamma \vdash_K^{\varepsilon} t : [a < 1] \cdot \sigma \multimap \tau \quad \phi; \Phi; \Delta \vdash_H^{\varepsilon} u : \sigma\{0/a\}}{\phi; \Phi; \Gamma \uplus \Delta \vdash_{K+H}^{\varepsilon} tu : \tau\{0/a\}} (App)$
$\frac{\phi; \Phi; \Gamma \vdash_M^{\varepsilon} t : \text{Nat}[J, K] \quad \phi; (J \leq 0, \Phi); \Delta \vdash_N^{\varepsilon} u : \tau \quad \phi; (K \geq 1, \Phi); \Delta \vdash_N^{\varepsilon} s : \tau}{\phi; \Phi; \Gamma \uplus \Delta \vdash_{M+N}^{\varepsilon} \text{ifz } t \text{ then } u \text{ else } s : \tau} (If)$	
$\frac{}{\phi; \Phi; \Gamma \vdash_0^{\varepsilon} \underline{n} : \text{Nat}[n, n]} (n)$	$\frac{\phi; \Phi; \Gamma \vdash_M^{\varepsilon} t : \text{Nat}[I, J]}{\phi; \Phi; \Gamma \vdash_M^{\varepsilon} s(t) : \text{Nat}[I + 1, J + 1]} (s)$
	$\frac{\phi; \Phi; \Gamma \vdash_M^{\varepsilon} t : \text{Nat}[I, J]}{\phi; \Phi; \Gamma \vdash_M^{\varepsilon} p(t) : \text{Nat}[I - 1, J - 1]} (p)$
$\frac{(b, \phi); (b < H, \Phi); \Gamma, x : [a < I] \cdot A \vdash_J^{\varepsilon} t : [a < 1] \cdot B \quad (a, b, \phi); (a < I, b < H, \Phi) \vdash_{\varepsilon} B\{0/a\} \{\bigotimes_b^{b+1, a} I + b + 1/b\} \sqsubseteq A}{\phi; \Phi; \sum_{b < H} \Gamma \vdash_{H + \sum_{b < H} J}^{\varepsilon} \text{fix } x. t : [a < K] \cdot B\{0/a\} \{\bigotimes_b^{0, a} I/b\}} (Fix)$	

Figure 3: Typing rules of  $\text{d}\ell\text{PCF}_V$ .

### 3.3 The Type System

**The Language of Types** The type system  $\text{d}\ell\text{PCF}_V$  can be seen as a refinement of PCF obtained by a linear decoration of its type derivations. *Linear* and *modal types* are defined as follows:

$$\begin{array}{ll}
A, B ::= & \sigma \multimap \tau & \text{linear types} \\
\sigma, \tau ::= & [a < I] \cdot A \mid \text{Nat}[I, J] & \text{modal types}
\end{array}$$

where  $I, J$  range over index terms and  $a$  ranges over index variables. Modal types need some comments. Natural numbers are freely duplicable, so  $\text{Nat}[I, J]$  is modal by definition. As a first approximation,  $[a < I] \cdot A$  can be thought of as a universal quantification of  $A$ , and so  $a$  is bound in the linear type  $A$ . Moreover, the condition  $a < I$  says that  $\sigma$  consists of all the instances of the linear type  $A$  where the variable  $a$  is successively instantiated with the values from 0 to  $I - 1$ , *i.e.*  $A\{0/a\}, \dots, A\{I - 1/a\}$ . For those readers who are familiar with linear logic, and in particular with BLL, the modal type  $[a < I] \cdot A$  is a generalisation of the BLL formula  $!_{a < p} A$  to arbitrary index terms. As such it can be thought of as representing the type  $A\{0/a\} \otimes \dots \otimes A\{I - 1/a\}$ .  $\text{Nat}[I]$  is syntactic sugar for  $\text{Nat}[I, I]$ . In the typing rules we are going to define, modal types need to be manipulated in an algebraic way. For this reason, two operations on modal types need to be introduced. The first one is a binary operation  $\uplus$  on modal types. Suppose that  $\sigma = [a < I] \cdot A\{a/c\}$  and that  $\tau = [b < J] \cdot A\{I + b/c\}$ . In other words,  $\sigma$  consists of the first  $I$  instances of  $A$ , *i.e.*  $A\{0/c\}, \dots, A\{I - 1/c\}$  while  $\tau$  consists of the next  $J$  instances of  $A$ , *i.e.*  $A\{I + 0/c\}, \dots, A\{I + J - 1/c\}$ . Their *sum*  $\sigma \uplus \tau$  is naturally defined as a modal type consisting of the first  $I + J$  instances of  $A$ , *i.e.*  $[c < I + J] \cdot A$ . Furthermore,  $\text{Nat}[I, J] \uplus \text{Nat}[I, J]$  is just  $\text{Nat}[I, J]$ . An operation of bounded sum on modal types can be defined by generalising the idea above: suppose that

$$\sigma = [b < J] \cdot A\{b + \sum_{d < a} J\{d/a\}/c\}.$$



$v$	$\star$	$\text{arg}(c, \pi)$	$>$	$c$	$\star$	$\text{fun}(v, \pi)$
$v$	$\star$	$\text{fun}(\langle \lambda x.t; \xi \rangle, \pi)$	$>$	$\langle t; (x \mapsto v) \cdot \xi \rangle$	$\star$	$\pi$
$v$	$\star$	$\text{fun}(\langle \text{fix } x.t; \xi \rangle, \pi)$	$>$	$\langle t; (x \mapsto \langle \text{fix } x.t; \xi \rangle) \cdot \xi \rangle$	$\star$	$\text{arg}(v, \pi)$
$\langle \underline{0}; \xi' \rangle$	$\star$	$\text{fork}(t, u, \xi, \pi)$	$>$	$\langle t; \xi \rangle$	$\star$	$\pi$
$\langle \underline{n+1}; \xi' \rangle$	$\star$	$\text{fork}(t, u, \xi, \pi)$	$>$	$\langle u; \xi \rangle$	$\star$	$\pi$
$\langle \underline{n}; \xi \rangle$	$\star$	$s(\pi)$	$>$	$\langle \underline{n+1}; \emptyset \rangle$	$\star$	$\pi$
$\langle \underline{n}; \xi \rangle$	$\star$	$p(\pi)$	$>$	$\langle \underline{n-1}; \emptyset \rangle$	$\star$	$\pi$

Figure 4:  $\text{CEK}_{\text{PCF}}$  evaluation rules for value closures.

Then its *bounded sum*  $\sum_{a < I} \sigma$  is just  $[c < \sum_{a < I} J] \cdot A$ . Finally,  $\sum_{a < I} \text{Nat}[J, K] = \text{Nat}[J, K]$ , provided  $a$  is not free in  $J$  nor in  $K$ .

**Subtyping** Central to  $d\ell\text{PCF}_V$  is the notion of subtyping. An inequality relation  $\sqsubseteq$  between (linear or modal) types can be defined using the formal system in Figure 2. This relation corresponds to lifting index inequalities at the type level. Please observe that  $\sqsubseteq$  is a pre-order, *i.e.*, a reflexive and transitive relation.

**Typing** A typing judgement is of the form

$$\phi; \Phi; \Gamma \vdash_K^{\mathcal{E}} t : \tau,$$

where  $K$  is the *weight* of  $t$ , that is (informally) the maximal number of substitutions involved in the CBV evaluation of  $t$ .  $\Phi$  is a set of constraints (*cf.* Section 3.1) that we call the *index context*, and  $\Gamma$  is a context assigning a modal type to (at least) each free variable of  $t$ . Both sums and bounded sums are naturally extended from modal types to contexts (with, for instance,  $\{x : \sigma; y : \tau\} \uplus \{x : \sigma', z : \tau'\} = \{x : \sigma \uplus \sigma'; y : \tau; z : \tau'\}$ ). There might be free index variables in  $\Phi, \Gamma, \tau$  and  $K$ , all of them from  $\phi$ . Typing judgements can be derived from the rules of Figure 3.

Derivation rules for abstractions and applications have been informally presented in Section 2.2. The other ones are then intuitive, except the derivation rule for typing  $\text{fix } x.t$ , that is worth an explanation: to simplify, assume we want to type only one copy of its type (that is,  $K = 1$ ). To compute the weight of  $\text{fix } x.t$ , we need to know the number of times  $t$  will be copied during the evaluation, that is the number of nodes in the tree of its recursive calls. This tree is described by  $I$  (as explained in Section 3.1), since each occurrence of  $x$  in  $t$  stands for a recursive call. It has, say,  $H = \bigtriangleup_b^{0,1} I$  nodes. At each node  $b$  of this tree, the  $a^{\text{th}}$  occurrence of  $x$  will be replaced by the  $a^{\text{th}}$  son of  $b$ , *i.e.* by  $b+1 + \bigtriangleup_b^{b+1,a} I$ . The types have to match, and that is what the second premise expresses. Finally, the type of  $\text{fix } x.t$  is the type of the “main” copy of  $t$ , at the root of the tree (*i.e.*, at  $b = 0$ ). The weight counts all the recursive calls (*i.e.*,  $H$ ) plus the weight of each copy of  $t$  (*i.e.*, the weight of  $t$  for each  $b < H$ ).

Last, the subsumption rule allows to relax the precision standards of a typing judgement. One can also restrict the inequalities on indexes to equalities in this rule, and thereby construct only *precise* typing judgements. Observe that the set of all rules but this one is syntax directed. Moreover the subsumption rule preserves the PCF skeleton of the types, and so the type system is itself syntax directed *up to* index inequalities.

### 3.4 An Abstract Machine for PCF

The call-by-value evaluation of PCF terms can be faithfully captured by an abstract machine in the style of CEK [12], which will be introduced in this section.

The internal state of the  $\text{CEK}_{\text{PCF}}$  machine consists of a closure and a stack, interacting following a set of rules. Formally, a *value closure* is a pair  $v = \langle v; \xi \rangle$  where  $v$  is a value and  $\xi$  is an *environment*, itself a list of assignments of value closures to variables:

$$\xi ::= \emptyset \mid (x \mapsto v) \cdot \xi.$$

$\langle x; \xi \rangle$	$\star$	$\pi$	$>$	$\xi(x)$	$\star$	$\pi$
$\langle tu; \xi \rangle$	$\star$	$\pi$	$>$	$\langle t; \xi \rangle$	$\star$	$\arg(\langle u; \xi \rangle, \pi)$
$\langle s(t); \xi \rangle$	$\star$	$\pi$	$>$	$\langle t; \xi \rangle$	$\star$	$s(\pi)$
$\langle p(t); \xi \rangle$	$\star$	$\pi$	$>$	$\langle t; \xi \rangle$	$\star$	$p(\pi)$
$\langle \text{ifz } t \text{ then } u \text{ else } s; \xi \rangle$	$\star$	$\pi$	$>$	$\langle t; \xi \rangle$	$\star$	$\text{fork}(u, s, \xi, \pi)$

Figure 5:  $\text{CEK}_{\text{PCF}}$  contextual evaluation rules.

A *closure* is a pair  $c = \langle t; \xi \rangle$  where  $t$  is a term (and not necessarily a value). *Stacks* are terms from the following grammar:

$$\begin{aligned} \pi ::= & \diamond \mid \text{fun}(v, \pi) \mid \arg(c, \pi) \\ & \mid \text{fork}(t, u, \xi, \pi) \mid s(\pi) \mid p(\pi). \end{aligned}$$

A *process*  $P$  is a pair  $c \star \pi$  of a closure and a stack.

Processes evolve according to a number of rules. Some of them (see Figure 4) describe how the  $\text{CEK}_{\text{PCF}}$  machine evolves when the first component of the process is a value closure. Other rules (see Figure 5) prescribe the evolution of  $\text{CEK}_{\text{PCF}}$  in all the other cases.

The following tells us that  $\text{CEK}_{\text{PCF}}$  is an adequate methodology to evaluate PCF terms:

**Proposition 3.1 (Adequacy)** *If  $t$  is a PCF term of type  $\text{Nat}$ , then  $t \rightarrow_v^* \underline{n}$  iff  $(\langle t; \emptyset \rangle \star \diamond) >^* (\langle \underline{n}; \emptyset \rangle \star \diamond)$ .*

**Weights and  $\text{CEK}_{\text{PCF}}$  Machine** As it will be formalised in Section 5.3, an upper bound for the evaluation of a given term in the  $\text{CEK}_{\text{PCF}}$  machine can be obtained by multiplying its weight and its size. This results can be explained as follows: we have seen (in Section 3.3) that its weight represents the maximal number of substitutions in its CBV evaluation, and thereby the maximal number of steps of the form

$$v \star \text{fun}(\langle \lambda x.t; \xi \rangle, \pi) > \langle t; (x \mapsto v) \cdot \xi \rangle \star \pi \quad (1)$$

$$v \star \text{fun}(\langle \text{fix } x.t; \xi \rangle, \pi) > \langle t; (x \mapsto \langle \text{fix } x.t; \xi \rangle) \cdot \xi \rangle \star \arg(v, \pi) \quad (2)$$

in its evaluation with the  $\text{CEK}_{\text{PCF}}$ . Between two such steps, the use of the other rules is not taken into account by the weight; however these other rules make the *size* of the process to decrease.

## 4 Examples

In this section we will see how to type some “real life” functions in  $d\ell\text{PCF}_V$ , and what is the cost associated to them.

**Addition** In PCF, addition can be computed as follows:

$$\text{add} = \text{fix } f.\lambda yz. \text{ifz } y \text{ then } z \text{ else } s(fp(y)z),$$

and has PCF type  $\text{Nat} \Rightarrow \text{Nat} \Rightarrow \text{Nat}$ . A brief analysis of its evaluation, if we apply it to two values  $v$  and  $w$  in  $\text{Nat}$ , indicates that a correct annotation for this type in  $d\ell\text{PCF}_V$  would be

$$[a < 1] \cdot (\text{Nat}[f] \multimap [c < 1] \cdot (\text{Nat}[g] \multimap \text{Nat}[f + g]))$$

where  $f$  and  $g$  are constant symbols representing the values of  $t$  and  $u$  respectively. Since we directly apply  $\text{add}$ , without copying this function, the index variables  $a$  and  $c$  are bounded with 1. This type is indeed derivable for  $\text{add}$  in  $d\ell\text{PCF}_V$ , assuming that the equational program  $\mathcal{E}$  is powerful

$$\begin{array}{c}
A = \text{Nat}[J] \multimap [c < 1] \cdot (\text{Nat}[g] \multimap \text{Nat}[J + g]) \quad ; \quad C = \text{Nat}[H] \multimap [c < 1] \cdot (\text{Nat}[g] \multimap \text{Nat}[H + g]) \\
\Gamma = \{x : [a < 1] \cdot A, y : \text{Nat}[H], z : \text{Nat}[g]\} \quad ; \quad \phi = \{b, a, c\} \quad ; \quad \Phi = \{b < f + 1, a < 1, c < 1\} \\
\\
\frac{\phi; (H \geq 1, \Phi); x : [a < 1] \cdot A \vdash_0^\varepsilon x : [a < 1] \cdot A \quad \phi; (H \geq 1, \Phi); y : \text{Nat}[H] \vdash_0^\varepsilon y : \text{Nat}[H] \quad (p)}{\phi; (H \geq 1, \Phi); y : \text{Nat}[H] \vdash_0^\varepsilon p(y) : \text{Nat}[J]} \quad (App) \\
\\
\frac{\phi; (H \geq 1, \Phi); x : [a < 1] \cdot A, y : \text{Nat}[H] \vdash_0^\varepsilon x \, p(y) : [c < 1] \cdot (\text{Nat}[g] \multimap \text{Nat}[J + g])}{\phi; (H \geq 1, \Phi); \Gamma \vdash_0^\varepsilon x \, p(y) z : \text{Nat}[J + g]} \quad (App) \\
\\
\frac{\vdots \quad \phi; (H \geq 1, \Phi); z : \text{Nat}[g] \vdash_0^\varepsilon z : \text{Nat}[g]}{\phi; (H \geq 1, \Phi); \Gamma \vdash_0^\varepsilon x \, p(y) z : \text{Nat}[J + g]} \quad (s) \\
\\
\frac{\phi; \Phi; y : \text{Nat}[H] \vdash_0^\varepsilon y : \text{Nat}[H] \quad \phi; (H \leq 0, \Phi); \Gamma \vdash_0^\varepsilon z : \text{Nat}[H + g] \quad \vdots}{\phi; \Phi; \Gamma \vdash_0^\varepsilon \text{ifz } y \text{ then } z \text{ else } s(x \, p(y) z) : \text{Nat}[H + g]} \quad (If) \\
\\
\frac{(b, a); (b < f + 1, a < 1); (x : [a < 1] \cdot A; y : \text{Nat}[H]) \vdash_1^\varepsilon \lambda z. \text{ifz } y \text{ then } z \text{ else } s(x \, p(y) z) : [c < 1] \cdot (\text{Nat}[g] \multimap \text{Nat}[H + g])}{(b, a); (b < f + 1, a < 1); (x : [a < 1] \cdot A \vdash_{1+1}^\varepsilon \lambda y z. \text{ifz } y \text{ then } z \text{ else } s(x \, p(y) z) : [a < 1] \cdot C \quad b; b < f + 1 \models_\varepsilon C\{b + 1/b\} \equiv A)} \quad (-\circ) \\
\\
\frac{(Fix) \quad b; b < f + 1; x : [a < 1] \cdot A \vdash_{1+1}^\varepsilon \lambda y z. \text{ifz } y \text{ then } z \text{ else } s(x \, p(y) z) : [a < 1] \cdot C \quad b; b < f + 1 \models_\varepsilon C\{b + 1/b\} \equiv A}{\vdash_{f+1+\sum_{b < f+1} (1+1)}^\varepsilon \text{add} : [a < 1] \cdot \text{Nat}[f] \multimap [c < 1] \cdot (\text{Nat}[g] \multimap \text{Nat}[f + g])} \quad (-\circ)
\end{array}$$

Figure 6: Typing derivation of add

enough to assign the following meaning to the corresponding index (they all depend on a free index variable  $b$ ):

$$\begin{aligned}
I &= \text{if } b < f \text{ then } 1 \text{ else } 0; \\
J &= f - b - 1; \\
H &= f - b; \\
K &= f - b + 1.
\end{aligned}$$

The derivation is given in Figure 6. We omit all the subsumption steps, but the index equalities they use are easy to check given that the number of nodes in the tree of recursive calls is  $\bigtriangleup_b^{0,1} I = f + 1$ . The final weight is equal to  $3 \times (f + 1)$ .

**Multiplication** The multiplication can be easily defined using the addition:

$$\text{mult} = \text{fix } x. \lambda y z. \text{ifz } y \text{ then } 0 \text{ else add } z \, (x \, p(y) z).$$

Taking the indexes  $I, J, H$  and  $K$  defined as in the previous paragraph, and using the typing judgement for **add** with  $f$  replaced by  $g$  and  $g$  replaced by  $J \times g$ , we can assign to **mult** the type

$$[a < 1] \cdot \text{Nat}[f] \multimap [c < 1] \cdot (\text{Nat}[g] \multimap \text{Nat}[f \times g])$$

(see Figure 7). The weight of **mult** is equal to  $3 \times (f + 1) + \sum_{b < f+1} M$ , where the meaning of  $M$  is “if  $b = f$  then 0 else  $3g + 1$ ”. Thus the execution of the application of **mult** to two integers  $\underline{n}$  and  $\underline{m}$  in the  $\text{CEK}_{\text{PCF}}$  machine is proportional to  $n \times m$ .

## 5 The Metatheory of $\text{d}\ell\text{PCF}_V$

In this section, some metatheoretical results about  $\text{d}\ell\text{PCF}_V$  will be presented. More specifically, type derivations are shown to be modifiable in many different ways, all of them leaving the underlying term unaltered. These manipulations, described in Section 5.1, form a basic toolkit which is essential to achieve the main results of this paper, namely intentional soundness and completeness (which are presented in Section 5.3 and Section 5.4). Types are preserved by call-by-value reduction, as proved in Section 5.2.

$(\star) : \phi; (H \geq 1, \Phi); \emptyset \vdash_{3 \times (g+1)}^{\mathcal{E}} \text{add} : [a < 1] \cdot \text{Nat}[g] \multimap [c < 1] \cdot (\text{Nat}[J \times g] \multimap \text{Nat}[g + J \times g])$ $A = \text{Nat}[J] \multimap [c < 1] \cdot (\text{Nat}[g] \multimap \text{Nat}[J \times g]) \quad ; \quad C = \text{Nat}[H] \multimap [c < 1] \cdot (\text{Nat}[g] \multimap \text{Nat}[H \times g])$ $\Gamma = \{x : [a < 1] \cdot A, y : \text{Nat}[H], z : \text{Nat}[g]\} \quad ; \quad \phi = \{b, a, c\} \quad ; \quad \Phi = \{b < f + 1, a < 1, c < 1\}$	
$\frac{\phi; (H \geq 1, \Phi); x : [a < 1] \cdot A \vdash_0^{\mathcal{E}} x : [a < 1] \cdot A \quad \phi; (H \geq 1, \Phi); y : \text{Nat}[H] \vdash_0^{\mathcal{E}} p(y) : \text{Nat}[J]}{\phi; (H \geq 1, \Phi); x : [a < 1] \cdot A, y : \text{Nat}[H] \vdash_0^{\mathcal{E}} x \, p(y) : [c < 1] \cdot (\text{Nat}[g] \multimap \text{Nat}[J \times g])} \quad (App)$	$\frac{\phi; (H \geq 1, \Phi); y : \text{Nat}[H] \vdash_0^{\mathcal{E}} y : \text{Nat}[H]}{\phi; (H \geq 1, \Phi); y : \text{Nat}[H] \vdash_0^{\mathcal{E}} p(y) : \text{Nat}[J]} \quad (p)$
$\frac{(\star) \quad \phi; (H \geq 1, \Phi); z : \text{Nat}[g] \vdash_0^{\mathcal{E}} z : \text{Nat}[g]}{\phi; (H \geq 1, \Phi); \Gamma \vdash_{3 \times (g+1)}^{\mathcal{E}} \text{add} \, z : [c < 1] \cdot (\text{Nat}[J \times g] \multimap \text{Nat}[g + J \times g])} \quad (App)$	$\frac{\vdots \quad \phi; (H \geq 1, \Phi); z : \text{Nat}[g] \vdash_0^{\mathcal{E}} z : \text{Nat}[g]}{\phi; (H \geq 1, \Phi); \Gamma \vdash_0^{\mathcal{E}} x \, p(y) \, z : \text{Nat}[J \times g]} \quad (App)$
$\frac{\phi; (H \geq 1, \Phi); \Gamma \vdash_{3 \times (g+1)}^{\mathcal{E}} \text{add} \, z \, (x \, p(y) \, z) : \text{Nat}[H \times g]}{\phi; \Phi; y : \text{Nat}[H] \vdash_0^{\mathcal{E}} y : \text{Nat}[H] \quad \phi; (H \leq 0, \Phi); \Gamma \vdash_0^{\mathcal{E}} \underline{0} : \text{Nat}[H \times g] \quad \vdots} \quad (If)$	
$\frac{\phi; \Phi; \Gamma \vdash_M^{\mathcal{E}} \text{ifz } y \text{ then } \underline{0} \text{ else add } (x \, p(y) \, z) \, z : \text{Nat}[H \times g]}{(b, a); (b < f + 1, a < 1); (x : [a < 1] \cdot A; y : \text{Nat}[H]) \vdash_{i+M}^{\mathcal{E}} \lambda z. \text{ifz } y \text{ then } \underline{0} \text{ else add } (x \, p(y) \, z) \, z : [c < 1] \cdot (\text{Nat}[g] \multimap \text{Nat}[H \times g])} \quad (-\circ)$	
$\frac{(b, a); (b < f + 1, a < 1); (x : [a < 1] \cdot A \vdash_{i+1+M}^{\mathcal{E}} \lambda y z. \text{ifz } y \text{ then } \underline{0} \text{ else add } (x \, p(y) \, z) \, z : [a < 1] \cdot C}{\vdash_{f+1+\sum_{b < f+1} (1+1+M)}^{\mathcal{E}} \text{mult} : [a < 1] \cdot \text{Nat}[f] \multimap [c < 1] \cdot (\text{Nat}[g] \multimap \text{Nat}[f \times g])} \quad (Fix)$	

Figure 7: Typing derivation of mult

$\frac{}{\phi; \Phi \vdash_0^{\mathcal{E}} \diamond : (\tau, \tau)} \quad \frac{\phi; \Phi \vdash_I^{\mathcal{E}} \pi : (\sigma, \tau) \quad \phi; \Phi \vdash_{\mathcal{E}} \sigma' \sqsubseteq \sigma \quad \phi; \Phi \vdash_{\mathcal{E}} \tau \sqsubseteq \tau' \quad \phi; \Phi \models_{\mathcal{E}} I \leq J}{\phi; \Phi \vdash_J^{\mathcal{E}} \pi : (\sigma', \tau')}$	
$\frac{\phi; \Phi \vdash_J^{\mathcal{E}} c : \sigma\{0/a\} \quad \phi; \Phi \vdash_K^{\mathcal{E}} \pi' : (\tau\{0/a\}, \tau')}{\phi; \Phi \vdash_{J+K}^{\mathcal{E}} \text{arg}(c, \pi') : ([a < 1] \cdot (\sigma \multimap \tau), \tau')}$	$\frac{\phi; \Phi \vdash_J^{\mathcal{E}} v : [a < 1] \cdot (\sigma \multimap \tau) \quad \phi; \Phi \vdash_K^{\mathcal{E}} \pi' : (\tau\{0/a\}, \tau')}{\phi; \Phi \vdash_{J+K}^{\mathcal{E}} \text{fun}(v, \pi') : (\sigma\{0/a\}, \tau')}$
$\frac{\phi; N = 0, \Phi \vdash_J^{\mathcal{E}} \langle t; \xi \rangle : \sigma \quad \phi; M \geq 1, \Phi \vdash_J^{\mathcal{E}} \langle u; \xi \rangle : \sigma \quad \phi; \Phi \vdash_K^{\mathcal{E}} \pi' : (\sigma, \tau)}{\phi; \Phi \vdash_{J+K}^{\mathcal{E}} \text{fork}(t, u, \xi, \pi') : (\text{Nat}[M, N], \tau)}$	
$\frac{\phi; \Phi \vdash_I^{\mathcal{E}} \pi : (\text{Nat}[M + 1, N + 1], \tau)}{\phi; \Phi \vdash_I^{\mathcal{E}} s(\pi) : (\text{Nat}[M, N], \tau)}$	$\frac{\phi; \Phi \vdash_I^{\mathcal{E}} \pi : (\text{Nat}[M - 1, N - 1], \tau)}{\phi; \Phi \vdash_I^{\mathcal{E}} p(\pi) : (\text{Nat}[M, N], \tau)}$

Figure 8: dℓPCF<sub>V</sub>: Lifting Typing to Stacks

## 5.1 Manipulating Type Derivations

First of all, the constraints  $\Phi$  in index, subtyping and typing judgements can be made stronger without altering the rest:

**Lemma 5.1 (Strengthening)** *If  $\phi; \Psi \models_{\mathcal{E}} \Phi$ , then the following implications hold:*

1. *If  $\phi; \Phi \models_{\mathcal{E}} I \leq J$ , then  $\phi; \Psi \models_{\mathcal{E}} I \leq J$ ;*
2. *If  $\phi; \Phi \vdash_{\mathcal{E}} \sigma \sqsubseteq \tau$ , then  $\phi; \Psi \vdash_{\mathcal{E}} \sigma \sqsubseteq \tau$ ;*
3. *If  $\phi; \Phi; \Gamma \vdash_I^{\mathcal{E}} t : \sigma$ , then  $\phi; \Psi; \Gamma \vdash_I^{\mathcal{E}} t : \sigma$ .*

**Proof.** Point 1. is a trivial consequence of transitivity of implication in logic. Point 2. can be proved by induction on the structure of the proof of  $\phi; \Phi \vdash_{\mathcal{E}} \sigma \sqsubseteq \tau$ , using point 1. Point 3. can be proved by induction on a proof of  $\phi; \Phi; \Gamma \vdash_I^{\mathcal{E}} t : \sigma$ , using points 1 and 3.  $\square$

Strengthening is quite intuitive: whatever appears on the right of  $\vdash_{\mathcal{E}}$  should hold for all values of the variables in  $\phi$  satisfying  $\Phi$ , so strengthening corresponds to making the judgement weaker.

Fresh term variables can be added to the context  $\Gamma$ , leaving the rest of the judgement unchanged:

**Lemma 5.2 (Context Weakening)**  *$\phi; \Phi; \Gamma \vdash_I^{\mathcal{E}} t : \tau$  implies  $\phi; \Phi; \Gamma, \Delta \vdash_I^{\mathcal{E}} t : \tau$ .*

**Proof.** Again, this is an induction on the structure of a derivation for  $\phi; \Phi; \Gamma \vdash_I^{\mathcal{E}} t : \tau$ .  $\square$

Another useful transformation on type derivations consists in substituting index variables for defined index terms.

**Lemma 5.3 (Index Substitution)** *If  $\phi; \Phi \models_{\mathcal{E}} I \Downarrow$ , then the following implications hold:*

1. *If  $(a, \phi); \Phi, \Psi \models_{\mathcal{E}} J \leq K$ , then  $\phi; \Phi, \Psi\{I/a\} \models_{\mathcal{E}} J\{I/a\} \leq K\{I/a\}$  ;*
2. *If  $(a, \phi); \Phi, \Psi \vdash_{\mathcal{E}} \sigma \sqsubseteq \tau$ , then  $\phi; \Phi, \Psi\{I/a\} \vdash_{\mathcal{E}} \sigma\{I/a\} \sqsubseteq \tau\{I/a\}$  ;*
3. *If  $(a, \phi); \Phi, \Psi; \Gamma \vdash_J^{\mathcal{E}} t : \sigma$ , then  $\phi; \Phi, \Psi\{I/a\}; \Gamma\{I/a\} \vdash_{J\{I/a\}}^{\mathcal{E}} t : \sigma\{I/a\}$  .*

**Proof.** 1. Assume that  $\phi; \Phi \models_{\mathcal{E}} I \Downarrow$  and  $(a, \phi); \Phi, \Psi \models_{\mathcal{E}} J \leq K$ , and let  $\rho$  be an assignment satisfying  $\Phi, \Psi\{I/a\}$ . In particular,  $\rho$  satisfies  $\Phi$ , thus  $\llbracket I \rrbracket_{\rho}^{\mathcal{E}}$  is defined, say equal to  $n$ . For any index  $H$ ,  $\llbracket H\{I/a\} \rrbracket_{\rho}^{\mathcal{E}} = \llbracket H \rrbracket_{\rho, a \mapsto n}^{\mathcal{E}}$ . Hence  $(\rho, a \mapsto n)$  satisfies  $\Phi, \Psi$ , and then it also satisfies  $J \leq K$ . So  $\llbracket J\{I/a\} \rrbracket_{\rho}^{\mathcal{E}} = \llbracket J \rrbracket_{\rho, a \mapsto n}^{\mathcal{E}} \leq \llbracket K \rrbracket_{\rho, a \mapsto n}^{\mathcal{E}} = \llbracket K\{I/a\} \rrbracket_{\rho}^{\mathcal{E}}$ , and  $\rho$  satisfies  $J\{I/a\} \leq K\{I/a\}$ . Thus  $\phi; \Phi, \Psi\{I/a\} \models_{\mathcal{E}} J\{I/a\} \leq K\{I/a\}$ .

2. By induction on the subtyping derivation, using 1.

3. By induction on the typing derivation, using 1 and 2.  $\square$

Observe that the only hypothesis is that  $\phi; \Phi \models_{\mathcal{E}} I \Downarrow$  (definition in Section 3.1): we do not require  $I$  to be a value of  $a$  that satisfies  $\Psi$ . If it does not the constraints in  $\Phi, \Psi\{I/a\}$  become inconsistent, and the obtained judgements are vacuous.

## 5.2 Subject Reduction

What we want to prove in this subsection is the following result:

**Proposition 5.4 (Subject Reduction)** *If  $t \rightarrow_v u$  and  $\phi; \Phi; \emptyset \vdash_M^{\mathcal{E}} t : \tau$ , then  $\phi; \Phi; \emptyset \vdash_M^{\mathcal{E}} u : \tau$ .*

Subject Reduction can be proved in a standard way, by going through a Substitution Lemma, which only needs to be proved when the term being substituted is a *value*. Preliminary to the Substitution Lemma are two auxiliary results stating that derivations giving types to values can, if certain conditions hold, be split into two, or put in parametric form:

**Lemma 5.5 (Splitting)** *If  $\phi; \Phi; \Gamma \vdash_M^{\mathcal{E}} v : \tau_1 \uplus \tau_2$ , then there exist two indexes  $N_1, N_2$ , and two contexts  $\Gamma_1, \Gamma_2$ , such that  $\phi; \Phi; \Gamma_i \vdash_{N_i}^{\mathcal{E}} v : \tau_i$ , and  $\phi; \Phi \models_{\mathcal{E}} N_1 + N_2 \leq M$  and  $\phi; \Phi \vdash_{\mathcal{E}} \Gamma \sqsubseteq \Gamma_1 \uplus \Gamma_2$ .*

**Proof.** If  $v$  is a primitive integer  $\underline{n}$ , the result is trivial as the only possible decomposition of a type for integers is  $\text{Nat}[I, J] = \text{Nat}[I, J] \uplus \text{Nat}[I, J]$ .

If  $v = \lambda x.t$ , then its typing judgement derives from

$$(a, \phi); (a < I, \Phi); \Delta, x : \sigma \vdash_K^\varepsilon t : \tau \quad (3)$$

$$\phi; \Phi \vdash_\varepsilon \Gamma \sqsubseteq \sum_{a < I} \Delta \quad (4)$$

with  $\tau_1 \uplus \tau_2 = [a < I] \cdot \sigma \multimap \tau$  and  $M = I + \sum_{a < I} K$ . Hence  $I = I_1 + I_2$ , and  $\tau_1 = [a < I_1] \cdot \sigma \multimap \tau$ , and  $\tau_2 = [a < I_2] \cdot \sigma\{I_1 + a/a\} \multimap \tau\{I_1 + a/a\}$ . Since  $(a, \phi); (a < I_1, \Phi) \models_\varepsilon (a < I, \Phi)$ , we can strength the hypothesis in (3) by Lemma 5.1 and derive

$$\frac{(a, \phi); (a < I_1, \Phi); \Delta, x : \sigma \vdash_K^\varepsilon t : \tau}{\phi; \Phi; \sum_{a < I_1} \Delta \vdash_{I_1 + \sum_{a < I_1} K}^\varepsilon \lambda x.t : [a < I_1] \cdot \sigma \multimap \tau}$$

On the other hand, we can substitute  $a$  with  $a + I_1$  in (3) by Lemma 5.3, and derive

$$\frac{(a, \phi); (a < I_2, \Phi); \Delta\{a + I_1/a\}, x : \sigma\{a + I_1/a\} \vdash_{K\{a + I_1/a\}}^\varepsilon t : \tau\{a + I_1/a\}}{\phi; \Phi; \sum_{a < I_2} \Delta\{a + I_1/a\} \vdash_{I_2 + \sum_{a < I_2} K\{a + I_1/a\}}^\varepsilon \lambda x.t : [a < I_2] \cdot \sigma\{a + I_1/a\} \multimap \tau\{a + I_1/a\}}$$

Hence we can conclude with  $\Gamma_1 = \sum_{a < I_1} \Delta$ ,  $\Gamma_2 = \sum_{a < I_2} \Delta\{a + I_1/a\}$ ,  $N_1 = I_1 + \sum_{a < I_1} K$  and  $N_2 = I_2 + \sum_{a < I_2} K\{a + I_1/a\}$ .

Now, if  $v = \text{fix } x.t$ , then its typing judgement derives from

$$(b, \phi); (b < H, \Phi); \Delta, x : [a < I] \cdot A \vdash_J^\varepsilon t : [a < 1] \cdot B \quad (5)$$

$$\phi; \Phi \models_\varepsilon H \geq \bigotimes_b^{0, K} I \quad (6)$$

$$(a, b, \phi); (a < I, b < H, \Phi) \vdash_\varepsilon B\{0/a\}\{\bigotimes_b^{b+1, a} I + b + 1/b\} \sqsubseteq A \quad (7)$$

$$(a, \phi); (a < K, \Phi) \vdash_\varepsilon B\{0/a\}\{\bigotimes_b^{0, a} I/b\} \sqsubseteq C \quad (8)$$

$$\phi; \Phi \vdash_\varepsilon \Gamma \sqsubseteq \sum_{b < H} \Delta \quad (9)$$

with  $\tau_1 \uplus \tau_2 = [a < K] \cdot C$ , and  $M = H + \sum_{b < H} J$ . Hence  $K = K_1 + K_2$ , with  $\tau_1 = [a < K_1] \cdot C$ , and  $\tau_2 = [a < K_2] \cdot C\{a + K_1/a\}$ . Let  $H_1 = \bigotimes_b^{0, K_1} I$  and  $H_2 = \bigotimes_b^{H_1, K_2} I$ . Then  $H_1 + H_2 = \bigotimes_b^{0, K} I$ , and  $H_2$  is also equal to  $\bigotimes_b^{0, K_2} I\{H_1 + b/b\}$ . Just like the previous case, we can strengthen the hypothesis in (5), (7) and (8) and derive

$$\frac{\begin{array}{l} (b, \phi); (b < H_1, \Phi); \Delta, x : [a < I] \cdot A \vdash_J^\varepsilon t : [a < 1] \cdot B \\ (a, b, \phi); (a < I, b < H_1, \Phi) \vdash_\varepsilon B\{0/a\}\{\bigotimes_b^{b+1, a} I + b + 1/b\} \sqsubseteq A \\ (a, \phi); (a < K_1, \Phi) \vdash_\varepsilon B\{0/a\}\{\bigotimes_b^{0, a} I/b\} \sqsubseteq C \end{array}}{\phi; \Phi; \sum_{b < H_1} \Delta \vdash_{H_1 + \sum_{b < H_1} J}^\varepsilon \text{fix } x.t : [a < K_1] \cdot C}$$

Moreover, if we substitute  $b$  with  $b + H_1$  in (7) and we strengthen the constraints (since (6) implies  $\phi; \Phi, b < H_2 \models_\varepsilon \Phi, b + H_1 < H$ ), we get

$$(a, b, \phi); (a < I, b < H_2, \Phi) \vdash_\varepsilon B\{0/a\}\{\bigotimes_b^{b+1, a} I + b + 1/b\}\{H_1 + b/b\} \sqsubseteq A\{H_1 + b/b\}.$$

But  $(\bigotimes_b^{b+1, a} I + b + 1)\{H_1 + b/b\} = \bigotimes_b^{H_1 + b + 1, a} I + H_1 + b + 1$  and  $\bigotimes_b^{H_1 + b + 1, a} I = \bigotimes_b^{b+1, a}(I\{H_1 + b/b\})$ . Hence  $B\{0/a\}\{\bigotimes_b^{b+1, a} I + b + 1/b\}\{H_1 + b/b\} = B\{H_1 + b/b\}\{0/a\}\{\bigotimes_b^{b+1, a}(I\{H_1 + b/b\}) + b + 1/b\}$ . In the same way we can substitute  $a$  with  $a + K_1$  in (8):

$$(a, \phi); (a < K_2, \Phi) \vdash_\varepsilon B\{0/a\}\{\bigotimes_b^{0, a + K_1} I/b\} \sqsubseteq C\{a + K_1/a\}$$

But  $\bigotimes_b^{0, a + K_1} I = H_1 + \bigotimes_b^{H_1, a} I = H_1 + \bigotimes_b^{0, a} I\{H_1 + b/b\}$ , and so  $B\{0/a\}\{\bigotimes_b^{0, a + K_1} I/b\}$  is equivalent to  $B\{H_1 + b/b\}\{0/a\}\{\bigotimes_b^{0, a} I\{H_1 + b/b\}/b\}$ . Finally, by substituting also  $b$  with  $b + H_1$  in (5) we can derive

$$\begin{array}{c}
(b, \phi); (b < H_2, \Phi); \Delta\{H_1 + b/b\}, x : ([a < I] \cdot A)\{H_1 + b/b\} \vdash_{J\{H_1 + b/b\}}^{\mathcal{E}} t : [a < 1] \cdot B\{H_1 + b/b\} \\
(a, b, \phi); (a < I, b < H_2, \Phi) \vdash_{\mathcal{E}} B\{H_1 + b/b\}\{0/a\}\{\bigotimes_b^{b+1, a}(I\{H_1 + b/b\}) + b + 1/b\} \sqsubseteq A\{H_1 + b/b\} \\
(a, \phi); (a < K_2, \Phi) \vdash_{\mathcal{E}} B\{H_1 + b/b\}\{0/a\}\{\bigotimes_b^{0, a}I\{H_1 + b/b\}/b\} \sqsubseteq C\{a + K_1/a\} \\
\hline
\phi; \Phi; \sum_{b < H_2} \Delta\{H_1 + b/b\} \vdash_{H_2 + \sum_{b < H_2} J\{H_1 + b/b\}}^{\mathcal{E}} \text{fix } x.t : [a < K_2] \cdot C\{a + K_1/a\}
\end{array}$$

So we can conclude with  $\Gamma_1 = \sum_{a < H_1} \Delta$ ,  $\Gamma_2 = \sum_{a < H_2} \Delta\{a + H_1/a\}$ ,  $N_1 = H_1 + \sum_{a < H_1} J$  and  $N_2 = H_2 + \sum_{a < H_2} J\{a + H_1/a\}$ .  $\square$

**Lemma 5.6 (Parametric Splitting)** *If  $\phi; \Phi; \Gamma \vdash_M^{\mathcal{E}} v : \sum_{c < J} \sigma$  is derivable, then there exist an index  $N$  and a context  $\Delta$  such that one can derive  $c, \phi; c < J, \Phi; \Delta \vdash_N^{\mathcal{E}} v : \sigma$ , and  $\phi; \Phi \models_{\mathcal{E}} \sum_{c < J} N \leq M$  and  $\phi; \Phi \vdash_{\mathcal{E}} \Gamma \sqsubseteq \sum_{c < J} \Delta$ .*

**Proof.** The proof uses the same technique as for Lemma 5.5. If  $v$  is a lambda abstraction or a fixpoint, then  $\sum_{c < J} \sigma$  is on the form  $[a < \sum_{c < J} L] \cdot C$ , where  $[a < L] \cdot C\{a + \sum_{c' < c} L\{c'/c\}/a\} = \sigma$ . Then the result also follows from Strengthening (Lemma 5.1) and Index Substitution (Lemma 5.3): for the lambda abstraction, substitute  $a$  with  $a + \sum_{c' < c} L\{c'/c\}$  in (3). For the fixpoint consider the index  $H'$  satisfying the equations  $H'\{0/c\} = \bigotimes_b^{0, L\{0/c\}} I$  and  $H'\{i + 1/c\} = \bigotimes_b^{H\{i/c\}, L\{i+1/c\}} I$ . Then substitute  $b$  with  $b + \sum_{c' < c} H'\{c'/c\}$  (and add the constraint  $c < J$  in the context) in (5) and (7), and substitute  $a$  with  $a + \sum_{c' < c} L\{c'/c\}$  in (8) to derive the result.  $\square$

One can easily realise *why* these results are crucial for subject reduction: whenever the substituted value flows through a type derivation, there are various places where its type changes, namely when it reaches instances of the typing rules (*App*), (*-o*), (*If*) and (*Rec*): in all these cases the type derivation for the value must be modified, and the splitting lemmas certify that this is possible. We can this way reach the key intermediate result:

**Lemma 5.7 (Substitution)** *If  $\phi; \Phi; \Gamma, x : \sigma \vdash_M^{\mathcal{E}} t : \tau$  and  $\phi; \Phi; \emptyset \vdash_N^{\mathcal{E}} v : \sigma$  are both derivable, then there is an index  $K$  such that  $\phi; \Phi; \Gamma \vdash_K^{\mathcal{E}} t[x := v] : \tau$  and  $\phi; \Phi \models_{\mathcal{E}} K \leq M + N$ .*

**Proof.** The proof goes by induction on the derivation of the judgement  $\phi; \Phi; \Gamma, x : \sigma \vdash_M^{\mathcal{E}} t : \tau$ , making intense use of Lemma 5.5 and Lemma 5.6.  $\square$

Given Lemma 5.7, proving Proposition 5.4 is routine: the only two nontrivial cases are those where the fired redex is a  $\beta$ -redex or the unfolding of a recursively-defined function, and both consist in a substitution. Observe how Subject Reduction already embeds a form of *extensional* soundness for  $\text{d}\ell\text{PCF}_V$ , since types are preserved by reduction. As an example, if one builds a type derivation for  $\vdash_1^{\mathcal{E}} t : \text{Nat}[2, 7]$ , then the normal form of  $t$  (if it exists) is guaranteed to be a constant between 2 and 7. Observe, on the other hand, that nothing is known about the *complexity* of the underlying computational process yet, since the weight  $I$  does not necessarily decrease along reduction. This is the topic of the following section.

### 5.3 Intentional Soundness

In this section, we prove the following result:

**Theorem 5.8 (Intensional soundness)** *For any term  $t$ , if*

$$\vdash_H^{\mathcal{E}} t : \text{Nat}[I, J]$$

*then  $t \Downarrow^n \underline{m}$  where  $n \leq |t| \cdot ([H]^{\mathcal{E}} + 1)$  and  $[I]^{\mathcal{E}} \leq m \leq [J]^{\mathcal{E}}$ .*

Roughly speaking, this means that  $\text{d}\ell\text{PCF}_V$  also gives us some sensible information about the time complexity of evaluating typable PCF programs. The path towards Theorem 5.8 is not too short:

<b>Syntactic size of terms:</b>	<b>Size of closures:</b> $ \langle t; \xi \rangle  = \ t\ $
$ \underline{n}  = 2$	<b>Size of processes:</b> $ c \star \pi  =  c  +  \pi $
$ \lambda x.t  =  t  + 2$	<b>Size of stacks:</b>
$ \text{fix } x.t  =  t  + 2$	$ \diamond  = 0$
$ x  = 2$	$ \text{fun}(v, \pi)  =  v  +  \pi $
$ tu  =  t  +  u  + 2$	$ \text{arg}(c, \pi)  =  c  +  \pi  + 1$
$ \mathbf{s}(t)  =  t  + 2$	$ \text{fork}(t, u, \xi, \pi)  = \ t\  + \ u\  +  \pi  + 1$
$ \mathbf{p}(t)  =  t  + 2$	$ \mathbf{s}(\pi)  =  \pi  + 1$
$ \text{ifz } t \text{ then } u \text{ else } s  =  t  +  u  +  s  + 2$	$ \mathbf{p}(\pi)  =  \pi  + 1$

Figure 9: Size of processes

it is necessary to lift  $\text{d}\ell\text{PCF}_V$  to a type system for closures, environments and processes, as defined in Section 3.4. Actually, the type system can be easily generalised to closures by the rule below:

$$\frac{\phi; \Phi; x_1 : \sigma_1, \dots, x_n : \sigma_n \vdash_K^\mathcal{E} t : \tau \quad \phi; \Phi \vdash_{J_i}^\mathcal{E} v_i : \sigma_i}{\phi; \Phi \vdash_{K + \sum_{1 \leq i \leq n} J_i}^\mathcal{E} \langle t; \{x_1 \mapsto v_1; \dots; x_n \mapsto v_n\} \rangle : \tau}$$

Lifting everything to stacks, on the other hand, requires more work, see Figure 8. We say that a stack  $\pi$  is  $(\phi; \Phi)$ -*acceptable* for  $\sigma$  with type  $\tau$  with cost  $I$  (notation:  $\phi; \Phi \vdash_I^\mathcal{E} \pi : (\sigma, \tau)$ ) when it interacts well with closures of type  $\sigma$  to product a process of type  $\tau$ . Indeed, a *process* can be typed as follows:

$$\frac{\phi; \Phi \vdash_J^\mathcal{E} \pi : (\sigma, \tau) \quad \phi; \Phi \vdash_K^\mathcal{E} c : \sigma}{\phi; \Phi \vdash_{J+K}^\mathcal{E} c \star \pi : \tau}$$

This way, also the notion of weight has been lifted to processes, with the hope of being able to show that it strictly decreases at every evaluation step. Apparently, this cannot be achieved in full: sometimes the weight of a process does not change, but in that case another parameter is guaranteed to decrease, namely the *process size*. The size  $|c \star \pi|$  of  $c \star \pi$ , is defined as  $|c| + |\pi|$ , where:

- The size  $|c|$  of a closure  $\langle t; \xi \rangle$  is the *multiplicative* size of  $t$  (cf. Section 3.2).
- The size of  $|\pi|$  is the sum of the sizes of all closures appearing in  $\pi$  plus the number of occurrences of symbols (different from  $\diamond$  and  $\text{fun}$ ) in  $\pi$ .

The formal definition of  $|c \star \pi|$  is given in Figure 9.

The size of a process decreases by any evaluation steps, except the two ones performing a substitution (1) and (2). However, these two reduction rules make the *weight* of a process decrease, as formalised by the following proposition. By the way, these are the cases in which a box is opened up in the underlying linear logic proof.

**Proposition 5.9 (Weighted Subject Reduction)** *Assume  $P > R$  and  $\phi; \Phi \vdash_I^\mathcal{E} P : \tau$ . Then  $\phi; \Phi \vdash_J^\mathcal{E} R : \tau$  and*

- *either  $\phi; \Phi \models_\mathcal{E} I = J$  and  $|P| > |R|$ ,*
- *or  $\phi; \Phi \models_\mathcal{E} I > J$  and  $|P| + |s| > |R|$ , where  $s$  is a term appearing in  $P$ .*

**Proof.** 1. If  $P > R$  with a non substitution rule (any rule of Figure 4 or Figure 5 except (1) and (2)), then it is easy to check that  $|P| > |R|$ . Moreover, in all these cases  $P$  and  $R$  have the same type and the same weight. We detail some cases:



- If  $P = v \star \arg(c, \pi) > c \star \text{fun}(v, \pi) = R$ , then the typing of  $P$  derives from

$$\begin{array}{c}
\phi; \Phi \vdash_{\mathcal{H}}^{\mathcal{E}} c : \sigma_0\{0/a\} \\
\phi; \Phi \vdash_{\mathcal{L}}^{\mathcal{E}} \pi' : (\tau_0\{0/a\}, \tau) \\
\phi; \Phi \vdash_{\mathcal{E}} \sigma \sqsubseteq [a < 1] \cdot (\sigma_0 \multimap \tau_0) \\
\phi; \Phi \models_{\mathcal{E}} J = H + L \\
\hline
\phi; \Phi \vdash_{\mathcal{J}}^{\mathcal{E}} \arg(c, \pi) : (\sigma, \tau) \qquad \phi; \Phi \vdash_{\mathcal{K}}^{\mathcal{E}} v : \sigma \\
\phi; \Phi \models_{\mathcal{E}} I = J + K \\
\hline
\phi; \Phi \vdash_{\mathcal{I}}^{\mathcal{E}} v \star \arg(c, \pi) : \tau
\end{array}$$

Hence since subtyping is derivable (Lemma ??) we can derive for  $R$ :

$$\begin{array}{c}
\phi; \Phi \vdash_{\mathcal{K}}^{\mathcal{E}} v : [a < 1] \cdot (\sigma_0 \multimap \tau_0) \\
\phi; \Phi \vdash_{\mathcal{L}}^{\mathcal{E}} \pi : (\tau_0\{0/a\}, \tau) \\
\hline
\phi; \Phi \vdash_{\mathcal{L}+K}^{\mathcal{E}} \text{fun}(v, \pi) : (\sigma_0\{0/a\}, \tau) \qquad \phi; \Phi \vdash_{\mathcal{H}}^{\mathcal{E}} c : \sigma_0\{0/a\} \\
\phi; \Phi \models_{\mathcal{E}} I = H + L + K \\
\hline
\phi; \Phi \vdash_{\mathcal{I}}^{\mathcal{E}} c \star \text{fun}(v, \pi) : \tau
\end{array}$$

- If  $P = \langle tu; \xi \rangle \star \pi > \langle t; \xi \rangle \star \arg(\langle u; \xi \rangle, \pi) = R$ , then the typing of  $P$  derives from

$$\begin{array}{c}
\phi; \Phi; x_1 : \mu_1, \dots, x_n : \mu_n \vdash_{\mathcal{K}}^{\mathcal{E}} t : [a < N] \cdot \kappa \multimap \eta \\
\phi; \Phi; x_1 : \eta_1, \dots, x_n : \eta_n \vdash_{\mathcal{H}}^{\mathcal{E}} u : \kappa\{0/a\} \\
\phi; \Phi \vdash_{\mathcal{E}} \sigma_i \sqsubseteq \mu_i \uplus \eta_i \\
\phi; \Phi \models_{\mathcal{E}} N \geq 1 \\
\phi; \Phi \vdash_{\mathcal{E}} \eta\{0/a\} \sqsubseteq \sigma \\
\hline
\phi; \Phi; x_1 : \sigma_1, \dots, x_n : \sigma_n \vdash_{\mathcal{H}+K}^{\mathcal{E}} tu : \sigma \qquad \phi; \Phi \vdash_{\mathcal{J}_i}^{\mathcal{E}} v_i : \sigma_i \\
\hline
\phi; \Phi \vdash_{\mathcal{H}+K+\sum_{i \leq n} \mathcal{J}_i}^{\mathcal{E}} \langle tu; \xi \rangle : \sigma \qquad \phi; \Phi \vdash_{\mathcal{J}}^{\mathcal{E}} \pi : (\sigma, \tau) \\
\phi; \Phi \models_{\mathcal{E}} I = J + H + K + \sum_i \mathcal{J}_i \\
\hline
\phi; \Phi \vdash_{\mathcal{I}}^{\mathcal{E}} \langle tu; \xi \rangle \star \pi : \tau
\end{array}$$

In particular, since subtyping is derivable,  $\phi; \Phi \vdash_{\mathcal{J}_i}^{\mathcal{E}} v_i : \mu_i \uplus \eta_i$  for each  $i$ . By Lemma 5.5 (that can be trivially extended to closures), it means that there are some  $< X_i, N_i$  such that

$$\begin{array}{c}
\phi; \Phi \vdash_{\mathcal{M}_i}^{\mathcal{E}} v_i : \mu_i \\
\phi; \Phi \vdash_{\mathcal{N}_i}^{\mathcal{E}} v_i : \eta_i \\
\phi; \Phi \models_{\mathcal{E}} \mathcal{M}_i + \mathcal{N}_i = \mathcal{J}_i
\end{array}$$

Hence both these judgements are derivable:

$$\begin{array}{c}
\phi; \Phi; x_1 : \mu_1, \dots, x_n : \mu_n \vdash_{\mathcal{K}}^{\mathcal{E}} t : [a < 1] \cdot \kappa \multimap \eta \qquad \phi; \Phi \vdash_{\mathcal{M}_i}^{\mathcal{E}} v_i : \mu_i \\
\hline
\phi; \Phi \vdash_{\mathcal{K}+\sum_{i \leq n} \mathcal{M}_i}^{\mathcal{E}} \langle t; \xi \rangle : [a < 1] \cdot \kappa \multimap \eta \\
\text{and} \quad \phi; \Phi; x_1 : \eta_1, \dots, x_n : \eta_n \vdash_{\mathcal{H}}^{\mathcal{E}} u : \kappa\{0/a\} \qquad \phi; \Phi \vdash_{\mathcal{N}_i}^{\mathcal{E}} v_i : \eta_i \\
\hline
\phi; \Phi \vdash_{\mathcal{H}+\sum_{i \leq n} \mathcal{N}_i}^{\mathcal{E}} \langle u; \xi \rangle : \kappa\{0/a\}
\end{array}$$

Hence we can derive the following typing judgement for  $R$  (notice that subtyping is derivable for the stacks, with contravariance in the first type):

$$\begin{array}{c}
\phi; \Phi \vdash_{\mathcal{K}+\sum_{i \leq n} \mathcal{M}_i}^{\mathcal{E}} \langle t; \xi \rangle : [a < 1] \cdot \kappa \multimap \eta \qquad \phi; \Phi \vdash_{\mathcal{H}+\sum_{i \leq n} \mathcal{N}_i}^{\mathcal{E}} \langle u; \xi \rangle : \kappa\{0/a\} \\
\phi; \Phi \vdash_{\mathcal{J}}^{\mathcal{E}} \pi : (\eta\{0/a\}, \tau) \\
\hline
\phi; \Phi \vdash_{\mathcal{J}+H+\sum_i \mathcal{N}_i}^{\mathcal{E}} \arg(\langle u; \xi \rangle, \pi) : ([a < 1] \cdot \kappa \multimap \eta, \tau) \\
\hline
\phi; \Phi \vdash_{\mathcal{I}}^{\mathcal{E}} \langle t; \xi \rangle \star \arg(\langle u; \xi \rangle, \pi) : \tau
\end{array}$$

2. If  $P > R$  with a substitution rule...

□

Splitting and parametric splitting play a crucial role here, once appropriately generalised to value closures.

Given Proposition 5.9, Theorem 5.8 is within reach: the natural number  $|s|$  in Proposition 5.9 cannot be greater than the size of the term  $t$  we start from, since the only “new” terms created along reduction are constants in the form  $\underline{n}$  (which have null size).

## 5.4 (Relative) Completeness

In this section, we will prove some results about the expressive power of  $d\ell\text{PCF}_V$ , seen as a tool to prove intentional (but also extensional) properties of PCF terms. Actually,  $d\ell\text{PCF}_V$  is extremely powerful: every first-order PCF program computing the function  $f : \mathbb{N} \rightarrow \mathbb{N}$  in a number of steps bounded by  $g : \mathbb{N} \rightarrow \mathbb{N}$  can be proved to enjoy these properties by way of  $d\ell\text{PCF}_V$ , provided two conditions are satisfied:

- On the one hand, the equational program  $\mathcal{E}$  needs to be *universal*, meaning that every partial recursive function is expressible by some index terms. This can be guaranteed, as an example, by the presence of a universal program in  $\mathcal{E}$ .
- On the other hand, all *true* statements in the form  $\phi; \Phi \models_{\mathcal{E}} I \leq J$  must be “available” in the type system for completeness to hold. In other words, one cannot assume that those judgements are derived in a given (recursively enumerable) formal system, because this would violate Gödel’s Incompleteness Theorem. In fact, ours are completeness theorems *relative* to an oracle for the truth of those assumptions, which is precisely what happens in Floyd-Hoare logics [6].

**PCF Typing** The first step towards completeness is quite easy: propositional type systems in the style of PCF for terms, closures, stacks and processes need to be introduced. All of them can be easily obtained by erasing the index information from  $d\ell\text{PCF}_V$ . As an example, the typing rule for the application looks like

$$\frac{\Gamma \vdash_{\text{PCF}} t : \alpha \Rightarrow \beta \quad \Gamma \vdash_{\text{PCF}} u : \alpha}{\Gamma \vdash_{\text{PCF}} tu : \beta}$$

while processes can be typed by the following rule

$$\frac{\vdash_{\text{PCF}} \pi : (\alpha, \beta) \quad \vdash_{\text{PCF}} c : \alpha}{\vdash_{\text{PCF}} c \star \pi : \beta}$$

Given any type  $\sigma$  (respectively any type derivation  $\delta$ ) of  $d\ell\text{PCF}_V$ , the PCF type (respectively, the PCF type derivation) obtained by erasing all the index information will be denoted by  $\langle\!\langle\sigma\!\rangle\!\rangle$  (respectively, by  $\langle\!\langle\delta\!\rangle\!\rangle$ ). Of course both terms and processes enjoy subject reduction theorems with respect to PCF typing, and their proofs are much simpler than those for  $d\ell\text{PCF}_V$ . As an example, given a type derivation  $\delta$  for  $\vdash_{\text{PCF}} P : \text{Nat}$  (we might write  $\delta \triangleright \vdash_{\text{PCF}} P : \text{Nat}$ ) and  $P > R$ , a type derivation  $\delta'$  for  $\vdash_{\text{PCF}} R : \text{Nat}$  can be easily built by manipulating in a standard way  $\delta$ ; we write  $\delta > \delta'$ .

**Weighted Subject Expansion** The key ingredient for completeness is a dualisation of Weighted Subject Reduction:

**Proposition 5.10 (Weighted Subject Expansion)** *Suppose that  $\delta \triangleright \vdash_{\text{PCF}} P : \alpha$ , that  $\delta > \delta'$ , and that  $\theta' \triangleright \phi; \Phi \vdash_I^{\mathcal{E}} R : \tau$  where  $\langle\!\langle\theta'\!\rangle\!\rangle = \delta'$ . Then there is*

$$\theta \triangleright \phi; \Phi \vdash_J^{\mathcal{E}} P : \tau$$

*with  $\langle\!\langle\theta\!\rangle\!\rangle = \delta$  and  $\phi; \Phi \models_{\mathcal{E}} J \leq I + 1$ . Moreover,  $\theta$  can be effectively computed from  $\delta$ ,  $\theta'$  and  $\delta'$ .*

Proving Proposition 5.10 requires a careful analysis of the evolution of the  $\text{CEK}_{\text{PCF}}$  machine, similarly to what happened for Weighted Subject *Reduction*. But while in the latter it is crucial to be able to (parametrically) *split* type derivations for terms (and thus closures), here we need to be able to *join* them:

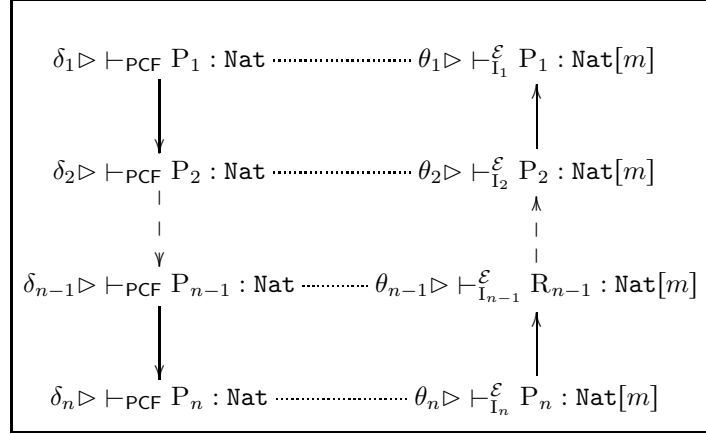


Figure 10: Completeness for Programs: sketch of the Proof

**Lemma 5.11 (Joining)** *If  $\mathcal{E}$  is universal, then*

$$\left. \begin{array}{l} \delta_i \triangleright \phi; \Phi; \Gamma_i \vdash_{N_i}^{\mathcal{E}} v : \tau_i \\ \langle \delta_1 \rangle = \langle \delta_2 \rangle \\ \phi; \Phi \vdash_{\mathcal{E}} \Gamma \sqsubseteq \Gamma_1 \uplus \Gamma_2 \\ \phi; \Phi \vdash_{\mathcal{E}} \tau_1 \uplus \tau_2 \sqsubseteq \tau \\ \phi; \Phi \models_{\mathcal{E}} N_1 + N_2 \leq M \end{array} \right\} \implies \phi; \Phi; \Gamma \vdash_M^{\mathcal{E}} v : \tau$$

**Lemma 5.12 (Parametric Joining)** *Suppose that  $\mathcal{E}$  is universal. Then*

$$\left. \begin{array}{l} a, \phi; a < I, \Phi; \Delta \vdash_N^{\mathcal{E}} v : \sigma \\ \phi; \Phi \vdash_{\mathcal{E}} \Gamma \sqsubseteq \sum_{a < I} \Delta \\ \phi; \Phi \vdash_{\mathcal{E}} \sum_{a < I} \sigma \sqsubseteq \tau \\ \phi; \Phi \models_{\mathcal{E}} \sum_{a < I} N \leq M \end{array} \right\} \implies \phi; \Phi; \Gamma \vdash_M^{\mathcal{E}} v : \tau$$

Observe that the Joining Lemma requires the two type derivations to be joined to have the same PCF “skeleton”. This is essential, because otherwise it would not be possible to unify them into one single type derivation.

**Completeness for Programs** We now have all the necessary ingredients to obtain a first completeness result, namely one about programs (which are terms of type  $\mathbf{Nat}$ ). Suppose that  $t$  is a PCF program such that  $t \rightarrow_v^* m$ , where  $m$  is a natural number. By Proposition 3.1, there is a sequence of processes

$$P_1 > P_2 > \dots > P_n,$$

where  $P_1 = \langle \langle t; \emptyset \rangle \star \diamond \rangle$  and  $P_n = \langle \langle m; \emptyset \rangle \star \diamond \rangle$ . Of course,  $\vdash P_i : \mathbf{Nat}$  for every  $i$ . For obvious reasons,  $\vdash_0^{\mathcal{E}} P_n : \mathbf{Nat}[m]$ . Moreover, by Weighted Subject Expansion, we can derive each of  $\vdash_{I_i}^{\mathcal{E}} P_i : \mathbf{Nat}[m]$ , until we reach  $\vdash_{I_1}^{\mathcal{E}} P_1 : \mathbf{Nat}[m]$ , where  $I_1 \leq n$  (see Figure 10 for a graphical representation of the above argument). It should be now clear that one can reach the following:

**Theorem 5.13 (Completeness for Programs)** *Suppose that  $\vdash_{\text{PCF}} t : \mathbf{Nat}$ , that  $t \Downarrow^n \underline{m}$  and that  $\mathcal{E}$  is universal. Then,  $\vdash_k^{\mathcal{E}} t : \mathbf{Nat}[m]$ , where  $k \leq n$ .*

**Uniformisation and Completeness for Functions** Completeness for programs, however, is not satisfactory: the fact (normalising) PCF terms of type  $\mathbf{Nat}$  can all be analysed by  $\text{d}\ell\text{PCF}_V$  is not so surprising, and other type systems (like non-idempotent intersection types [11]) have comparable expressive power. Suppose we want to generalise relative completeness to first-order functions: we would like to prove that every term  $t$  having a PCF type  $\mathbf{Nat} \Rightarrow \mathbf{Nat}$  (which terminates

when fed with any natural number) can be typed in  $\text{d}\ell\text{PCF}_V$ . How could we proceed? First of all, observe that the argument in Figure 10 could be applied to all *instances* of  $t$ , namely to all terms in  $\{tn \mid n \in \mathbb{N}\}$ . This way one can obtain, for every  $n \in \mathbb{N}$ , a type derivation  $\delta_n$  of

$$\vdash_{I_n}^{\mathcal{E}} t : [a < J_n] \cdot \text{Nat}[K_n] \multimap \text{Nat}[H_n]$$

where  $J_n$  can be assumed to be 1, while  $K_n$  can be assumed to be  $n$ . Moreover, the problem of obtaining  $\delta_n$  from  $n$  is recursive, i.e., can be solved by an algorithm. Surprisingly, the infinitely many type derivations in  $\{\delta_n \mid n \in \mathbb{N}\}$  can be turned into one:

**Proposition 5.14 (Uniformisation of type derivations)** *Suppose that  $\mathcal{E}$  is universal and that  $\{\delta_n\}_{n \in \mathbb{N}}$  is a recursively enumerable class of type derivations satisfying the following constraints:*

1. *For every  $n \in \mathbb{N}$ ,  $\theta_n \triangleright \vdash_{I_n}^{\mathcal{E}} t : \sigma_n$ ;*
2. *all derivations have the same skeleton*

*Then there is a type derivation  $\theta \triangleright a; \emptyset; \emptyset \vdash_I^{\mathcal{E}} t : \sigma$  such that  $\models_{\mathcal{E}} I\{n/a\} = I_n$  and  $\models_{\mathcal{E}} \sigma\{n/a\} \equiv \sigma_n$  for all  $n$ .*

Uniformisation of type derivations should be seen as an extreme form of joining: not only a finite number of type derivations for the same term can be unified into one, but even any recursively enumerable class of them can. Again, the universality of  $\mathcal{E}$  is crucial here. We are now ready to give the following:

**Theorem 5.15 (Completeness for functions)** *Suppose that  $\vdash_{\text{PCF}} t : \text{Nat} \Rightarrow \text{Nat}$ , that  $t \underline{n} \Downarrow^{k_n} \underline{m}_n$  for all  $n \in \mathbb{N}$  and that  $\mathcal{E}$  is universal. Then, there is an index  $H$  such that  $a; \emptyset; \emptyset \vdash_I^{\mathcal{E}} t : [b < 1] \cdot \text{Nat}[a] \multimap \text{Nat}[H]$ , where  $\models_{\mathcal{E}} I\{n/a\} \leq k_n$  and  $\models_{\mathcal{E}} H\{n/a\} = m_n$ .*

## 6 Further Developments

Relative completeness of  $\text{d}\ell\text{PCF}_V$ , especially in its stronger form (Theorem 5.15) can be read as follows. Suppose that a (sound), finitary formal  $\mathcal{C}$  system deriving judgements in the form  $\phi; \Phi \vdash_{\mathcal{E}} I \leq J$  is fixed and “plugged” into  $\text{d}\ell\text{PCF}_V$ . What you obtain is a sound, but necessarily incomplete formal system, due to Gödel’s incompleteness. However, this incompleteness is *only* due to  $\mathcal{C}$  and not to the rules of  $\text{d}\ell\text{PCF}_V$ , which are designed so as to reduce the problem of proving properties of programs to checking inequalities over  $\mathcal{E}$  *without any loss of information*.

In this scenario, it is of paramount importance to devise techniques to *automatically* reduce the problem of checking whether a program satisfies a given intentional or extensional specification to the problem of checking whether a given set of inequalities over an equational program  $\mathcal{E}$  hold. Indeed, many techniques and concrete tools are available for the latter problem (take, as an example, the immense literature on SMT solving), while the same cannot be said about the former problem. The situation, in a sense, is similar to the one in the realm of program logics for imperative programs, where logics are indeed very powerful [6], and great effort have been directed to devise efficient algorithms generating weakest preconditions [10].

Actually, at the time of writing, the authors are actively involved in the development of *relative type inference* algorithms for both  $\text{d}\ell\text{PCF}_N$  and  $\text{d}\ell\text{PCF}_V$ , which can be seen as having the same role as algorithms computing weakest preconditions. This is however out of the scope of this paper.

## 7 Conclusions

Linear dependent types are shown to be applicable to the analysis of intentional and extensional properties of functional programs when the latter are call-by-value evaluated. More specifically, soundness and relative completeness results are proved for both programs and functions. This generalises previous work by Gaboardi and the first author [8], who proved similar results in

the call-by-name setting. This shows that linear dependency not only provides an expressive formalism, but is also robust enough to be adaptable to calculi whose notions of reduction are significantly different (and more efficient) than normal order evaluation.

Topics for future work include some further analysis about the applicability of linear dependent types to languages with more features, including some form of inductive data types, or ground type references.

## References

- [1] A. Asperti and H. G. Mairson. Parallel beta reduction is not elementary recursive. *Inf. Comput.*, 170(1):49–80, 2001.
- [2] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.
- [3] P. Baillot, M. Gaboardi, and V. Mogbil. A polytime functional language from light linear logic. In *ESOP*, volume 6012 of *LNCS*, pages 104–124. Springer, 2010.
- [4] P. Baillot and K. Terui. Light types for polynomial time computation in lambda calculus. *J. & C.*, 207(1):41–62, 2009.
- [5] G. Barthe, B. Grégoire, and C. Riba. Type-based termination with sized products. In *CSL*, volume 5213 of *LNCS*, pages 493–507. Springer, 2008.
- [6] S. A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM J. on Computing*, 7:70–90, 1978.
- [7] U. Dal Lago. Context semantics, linear logic, and computational complexity. *ACM Trans. Comput. Log.*, 10(4), 2009.
- [8] U. Dal Lago and M. Gaboardi. Linear dependent types and relative completeness. In *LICS*, pages 133–142, 2011.
- [9] U. Dal Lago and B. Petit. Linear dependent types in a call-by-value scenario. available at <http://www.cs.unibo.it/~dallago/ldtcbv.pdf>, 2012.
- [10] J. W. de Bakker, A. de Bruin, and J. Zucker. *Mathematical theory of program correctness*. Prentice-Hall international series in computer science. Prentice Hall, 1980.
- [11] D. de Carvalho. Execution time of lambda-terms via denotational semantics and intersection types. available at <http://arxiv.org/abs/0905.4251>, 2009.
- [12] M. Felleisen and D. P. Friedman. Control operators, the SECD-machine and the  $\lambda$ -calculus. Technical Report 197, Computer Science Department, Indiana University, 1986.
- [13] J.-Y. Girard, A. Scedrov, and P. J. Scott. Bounded linear logic: A modular approach to polynomial-time computability. *Theor. Comput. Sci.*, 97(1):1–66, 1992.
- [14] J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate Amortized Resource Analysis. In *ACM POPL*, pages 357–370, 2011.
- [15] M. Hofmann. Linear types and non-size-increasing polynomial time computation. In *LICS*, pages 464–473. IEEE Comp. Soc., 1999.
- [16] S. Jost, K. Hammond, H.-W. Loid, and M. Hofmann. Static Determination of Quantitative Resource Usage for Higher-Order Programs. In *ACM POPL*, Madrid, Spain, 2010.
- [17] N. Kobayashi and C.-H. L. Ong. A type system equivalent to the modal mu-calculus model checking of higher-order recursion schemes. In *LICS*, pages 179–188. IEEE Comp. Soc., 2009.

- [18] J. Lamping. An algorithm for optimal lambda calculus reduction. In *POPL*, pages 16–30. ACM Press, 1990.
- [19] J. Maraist, M. Odersky, D. N. Turner, and P. Wadler. Call-by-name, call-by-value, call-by-need and the linear lambda calculus. *Electr. Notes Theor. Comput. Sci.*, 1:370–392, 1995.
- [20] P. Odifreddi. *Classical Recursion Theory: the Theory of Functions and Sets of Natural Numbers*. Number 125 in Studies in Logic and the Foundations of Mathematics. North-Holland, 1989.
- [21] G. D. Plotkin. LCF considered as a programming language. *Theor. Comp. Sci.*, 5:225–255, 1977.
- [22] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE JSAC*, 21(1):5–19, 2003.
- [23] D. M. Volpano, C. E. Irvine, and G. Smith. A sound type system for secure flow analysis. *JCS*, 4(2/3):167–188, 1996.
- [24] H. Xi. Dependent types for program termination verification. In *LICS*, pages 231–246. IEEE Comp. Soc., 2001.